Programming Python

Second Edition

Mark Lutz

O'REILLY®

Программирование на Python

Второе издание

Марк Лутц

Санкт-Петербург
2002

Марк Лутц

Программирование на Python, 2 издание

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*М. Деркачев*
Редактор	*А. Лосев*
Корректура	*С. Беляева*
Верстка	*Н. Гриценко*

Лутц М.
Программирование на Python. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 1136 с., ил.
ISBN 5-93286-036-7

Python – это широко распространенный язык программирования, применяемый при решении многих важных задач, диапазон которых простирается от коммерческих сценариев установки Linux и программирования веб-приложений до анимации фильмов и создания спецэффектов. Он доступен на всех ведущих вычислительных платформах, в том числе на основных коммерческих версиях Unix, Linux, Windows и Mac OS. Кроме того, он является языком с открытым исходным кодом.

Второе издание самого известного бестселлера по Python, прорецензированное и одобренное Гвидо ван Россумом, создателем Python, представляет собой наиболее полный на сегодняшний день источник для серьезно программирующих на Python. Основное внимание здесь сосредоточено на практическом применении языка. Читатель обнаружит, что одна книга фактически содержит в себе четыре, которые глубоко освещают создание сценариев для Интернета, системное программирование, программирование GUI с использованием Tkinter и интеграцию с C. Кроме того, обсуждаются новые инструменты и приложения: Jython – версия Python, компилируемая в виде байт-кодов Java; расширения Active Scripting и COM; Zope – система веб-приложений с открытым исходным кодом; генераторы кода HTMLgen и SWIG; поддержка потоков; модули CGI и протоколов Интернета. В книге приводится большое количество примеров кода, которые вы сможете использовать при разработке на Python сложных приложений. Прилагается CD для платформ PC, Macintoch и Unix.

ISBN 5-93286-036-7
ISBN 0-596-00085-5 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 18.09.2002. Формат 70x100 1/16 . Печать офсетная.
Объем 71 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Вступительное слово

Менее пяти лет назад я написал вступительное слово к первому изданию «Программирования на Python». За истекшее время книга изменилась примерно в той же степени, что и сам язык и сообщество Python! Мне больше не нужно защищать Python: статистика и события, приведенные Марком в предисловии, говорят сами за себя.

За последний год Python быстро развивался. Мы выпустили Python 2.0 – большой шаг вперед – с новыми возможностями, включенными в стандартную библиотеку, такими как поддержка Unicode и XML, и несколькими новыми синтаксическими конструкциями, в том числе улучшенным присвоением: теперь вместо `x = x+1` можно писать `x += 1`. Кое-кто считал, что это мелочь (представьте себе, однако, не `x`, а `dict[key]` или `list[index]`), но в целом это имело успех у тех пользователей, которые привыкли к использованию улучшенного присвоения в других языках.

Менее теплый прием получило расширение команды print, `print>>file` – сокращенная форма для вывода в файл, отличный от стандартного вывода. Лично я использую эту возможность Python 2.0 очень часто, но большинство высказавшихся по этому поводу считает ее отвратительной. Обсуждение данного простого расширения языка в телеконференции было одним из самых длинных в истории, исключая вечную тему «Python против Perl».

И это подсказывает следующую тему. (Нет, не противостояние Python и Perl. Предисловие – не место для стычек.) Я имею в виду скорость, с которой происходит развитие Python, – эту тему автор книги принимает близко к сердцу. Всякий раз, когда я ввожу в Python новую функцию, у Марка появляется еще одна седая прядь в волосах – еще одна глава устарела! Особое его беспокойство вызвало появление в Python 2.0 множества новых функций, добавленных как раз в то время, когда он работал над данным, вторым изданием книги. Что если в Python 2.1 появится столько же нововведений? Книга устареет в тот день, когда будет опубликована!

Марк, расслабься. Python продолжит свое развитие, но я обещаю, что те вещи, которые активно используются, сохранятся! Например, много беспокойства вызвал модуль для работы со строками. Сейчас, с появлением методов у объектов строк, этот модуль стал в значительной мере избыточным. Хотелось бы мне объявить его устаревшим и тем самым побудить программистов Python начать использовать вместо него методы строк. Но с учетом того, что большая часть существующего кода Python и даже многие стандартные модули используют модуль строк, такая перемена не может, очевидно, произойти очень быстро. Скорее всего, модуль строк мы сможем убрать не раньше выхода в свет Python 3000, и даже тогда, вероятно, этот модуль сохранится в библиотеке для поддержки обратной совместимости, чтобы иметь возможность использовать старый код.

Python 3000?! Да, это условное название интерпретатора Python следующего поколения. Название можно считать игрой слов с Windows 2000 или ассоциацией с «Mystery Science Theater 3000» – ТВ-шоу вполне в духе Python, ставшее культовым. Когда выйдет Python 3000? О-о-о-чень нескоро, хотя не обязательно придется ждать 3000-го года.

Первоначально Python 3000 должен был быть спроектирован и переписан заново. Это позволило бы мне осуществить несовместимые изменения, чтобы исправить те проблемы архитектуры языка, которые нельзя решить с сохранением обратной совместимости. В настоящее время, однако, планируется постепенное введение необходимых изменений в рамках продолжения текущей линии Python 2.x с ясным переходным периодом, в течение которого поддерживается обратная совместимость.

Возьмем, к примеру, деление целых чисел. Так же как и в C, в настоящее время в Python деление x/y для двух целочисленных аргументов имеет целочисленный результат. Иными словами, 1/2 дает 0! Для закоренелых программистов это естественно, но для новичков, составляющих все большую часть пользователей Python (количество которых растет экспоненциально), это служит беспрестанным источником недоразумений. С точки зрения вычислений более разумно, чтобы оператор / давал одно и то же значение независимо от типа его операндов: в конце концов, так поступают все числовые операторы. Но нельзя просто взять и изменить Python, чтобы 1/2 давало 0.5, потому что (как и в случае удаления модуля string) слишком большая часть существующего кода стала бы неработоспособной. Где же выход?

Решение – слишком сложное, чтобы описывать его здесь подробно, – охватит несколько последующих версий Python и состоит в постепенном усилении давления на программистов Python (сначала в документации, затем через предупреждения об использовании устаревших конструкций и, наконец, через сообщения об ошибках) с целью заставить их переработать свой код. Кстати, структура вывода предупредительных сообщений будет включена в версию Python 2.1. Прости, Марк!

Поэтому не ожидайте в скором времени объявления о выходе Python 3000. Вместо этого вы можете в один прекрасный день обнаружить, что *уже* используете Python 3000, только название его будет не таким, а скажем, чем-нибудь вроде Python 2.8.7. И большая часть того, что вы узнаете из этой книги, будет по-прежнему действовать! Тем не менее, в ссылках на Python 3000 не будет недостатка: просто помните, что это намеренно превозносимая химера в буквальном смысле слова. Вместо беспокойства по поводу Python 3000 лучше продолжайте использовать и изучать ту версию Python, которая у вас есть.

Хотелось бы сказать несколько слов о существующей в данное время модели разработки Python. До начала 2000 года разработчиков Python были сотни, но, в сущности, все нововведения проходили через мой почтовый ящик. Предлагая внести в Python изменение, вы должны были прислать мне по почте файл различий, который я применял к своей рабочей версии Python, и если изменение меня удовлетворяло, я вносил его в свое дерево исходного кода CVS. (CVS – это система управления версиями кода, которой посвящено несколько книг.) Сообщения о выявленных ошибках следовали тем же путем, за исключением того, что в итоге я выпускал заплатку. Очевидно, что с ростом числа предлагаемых изменений мой почтовый ящик стал узким местом процесса. Что было делать?

К счастью, Python – не единственный проект open source, в котором возникла эта проблема, и несколько умных людей в VA Linux предложили свое решение: SourceForge! Это динамичный веб-сайт с полным набором средств управления распределенными проектами: открытое хранилище CVS, почтовые списки рассылки (использующие Mailman, очень популярное приложение Python!), дискуссионные форумы, средства администрирования для ошибок и патчей и область для загрузки с сервера – все доступно для любого проекта open source, которому это может понадобиться.

В данное время у нас есть группа из 30 добровольцев с правами записи в SourceForge и почтовый список рассылки для разработчиков, в котором вдвое больше народа. Все привилегированные добровольцы поклялись на верность BDFL (Benevolent Dictator For Life – Великодушному Пожизненному Диктатору, то есть мне :-). Введение существенных новых функций регулируется с помощью облегченной системы предложений и обратной связи под названием Python Enhancement Proposals (PEPs). Наша система PEP оказалась столь успешной, что почти буквально была скопирована сообществом Tcl, когда оно совершило аналогичный переход с Cathedral на Bazaar.

Итак, с уверенностью в будущем Python я передаю слово Марку Лутцу. Отличная работа, Марк! Завершаю моей любимой цитатой из Monty Python: «Take it away, Eric, the orchestra leader»!

Гвидо ван Россум

Рестон, Вирджиния, январь 2001

Предисловие ко второму изданию

«А теперь нечто совершенно новое... опять»

Предыдущее издание этой книги стало одним из первых, представивших язык программирования Python. Второе издание является фактически новой книгой, посвященной более сложным темам Python, и должно послужить продолжением изложения основ языка в «Learning Python», дополненного справочными материалами из «Python Pocket Reference».

Это означает, что настоящее издание посвящено скорее возможностям *использования* Python, чем самому языку. Попутно исследуются идеи, лежащие в основе разработки программного обеспечения на Python; в действительности они обретают смысл только в контексте более крупных примеров, которые включены в данное издание. Но в целом, предполагается наличие у читателя хотя бы беглого знакомства с основами языка Python, которые помогут перейти к оставшейся части рассказа о нем.

В данном предисловии будут объяснены некоторые причины такой существенной переработки текста, более подробно описана структура издания и дан краткий обзор возможностей использования программ Python, имеющихся на прилагаемом компакт-диске. Однако вначале нам предстоит осуществить исторический экскурс.

Вехи распространения Python

Последние пять лет стали примечательными в мире Python. С тех пор как я написал первое издание данной книги в период между 1995 и 1996 годами, Python из новичка в семействе языков сценариев превратился в солидный и широко распространенный инструмент, используемый компаниями, разбросанными по всему свету. Хотя не всегда легко оценить популярность свободно распространяемого программного обеспечения с открытым кодом (*http://opensource.org*), такого как Python, имеющаяся статистика показывает экспоненциальный рост его популярности за последние пять лет. Вот некоторые из самых свежих признаков взрывного роста интереса к нему:

Книги

В 2001 году, когда пишется эта книга, в продаже имеется свыше десятка книг, посвященных Python, и еще почти столько же готовится к изданию (в 1995 книг не было). Некоторые из этих книг посвящены определенным областям (например, Windows), а некоторые изданы на немецком, французском и японском языках.

Пользователи

В 1999 году один из ведущих обозревателей индустрии программного обеспечения предположил исходя из различных статистических данных, что в мире насчитывается 300 000 пользователей Python. Другие оценки являются еще более оптимистичными. В начале 2000 года, например, уже действовал веб-сайт Python, с которого к концу года было произведено 500 000 новых загрузок интерпретатора (помимо того, что есть и другие носители дистрибутивов Python). Вероятно, на момент написания данной книги последняя цифра ближе к подлинной численности контингента пользователей.

Периодическая печать

Сейчас Python регулярно служит темой публикаций в периодических изданиях по программированию. В действительности после 1995 года создатель Python Гвидо ван Россум (Guido van Rossum) появился на обложке известных специальных журналов, например *Linux Journal* и *Dr. Dobb's Journal*; в публикации последнего создание Python стало основанием для награды за выдающиеся успехи в программировании.[1]

Приложения

Настоящие компании применяют Python для создания настоящих продуктов. Он был использован в построении спецэффектов для последнего фильма «Звездные войны» (Industrial Light & Magic), при выводе карт и каталогов в Интернете (Yahoo), в инсталляционной программе операционной системы Linux (Red Hat), при проверке микросхем и плат (Intel), в управлении дискуссионными форумами Интернета (Egroups), в сценариях сетевых игр (Origin), в интерфейсе к серверу CORBA (TCSI), в реализации инструментария веб-сайтов (Digital Creations' Zope), в сценариях для беспроводных устройств (Agilent) и во многих других местах.[2]

Телеконференции

Объем переписки в главной телеконференции по Python, *comp.lang.python*, также значительно вырос. Например, согласно eGroups (см. *http://www.egroups.com/group/python-list*), в январе 1994 года в этот список было подано 76 сообщений против 2678 в январе 2000 года – 35-кратное увеличение. В последующее время активность еще более возросла (4226 сообщений только за июнь 2000 – примерно 140 в сутки), и наблюдается постоянный рост с момента начала работы списка. К тому времени, когда вы будете читать это, цифры, относящиеся к числу пользователей и приведенные в данном предисловии, вероятно, возрастут. Но даже при настоящей интенсивности обмена форумы по Python достаточно загружены, чтобы полностью занять время того, кто готов полностью посвятить себя им.

Конференции

Сейчас ежегодно проводятся две конференции по Python, одну из которых организует O'Reilly & Associates. Число участников конференций, посвященных Python, примерно удваивалось каждый год. Кроме того, в Европе теперь проводится ежегодный «День Python».

Групповая терапия

Региональные группы пользователей Python стали возникать во многих местах в США и за границей, в том числе в Орегоне, Сан-Франциско, Вашингтоне, Италии, Корее и Англии. Такие группы работают над усовершенствованиями языка, проводят общественные мероприятия по Python и делают многое другое.

---

[1] Когда я писал эту книгу, *Linux Journal* напечатал также специальное приложение, посвященное Python, к майскому изданию 2000 года, на обложке которого, конечно, красовался голый мужчина, сидящий на улице перед компьютером вместо пианино. Если вам не понятно, почему это должно быть смешно, нужно посмотреть повторные показы телесериала «Monty Python», от которого получил свое название Python (считайте это первым предложенным упражнением). Я подробнее расскажу о последствиях, к которым привело название Python, в первой главе.

[2] Подробнее см. на *http://www.python.org*. Некоторые компании скрывают использование ими Python по соображениям конкуренции, хотя для многих это в итоге выявляется, когда происходит отказ на какой-либо из веб-страниц и в броузере появляется сообщение Python об ошибке. Обычно среди компаний, «засветившихся» подобным образом, упоминается Hewlett Packard.

Области применения

Развитие Python объединило как разработчиков для Microsoft Windows, с включением поддержки COM и Active Scripting, так и разработчиков Java благодаря новой, основанной на Java реализации языка – JPython (переименованного в «Jython»). Как будет показано в данной книге, появившаяся в языке поддержка COM позволяет сценариям Python являться как серверами компонентов, так и клиентами этих серверов; Active Scripting позволяет встраивать код Python в страницы HTML и выполнять его у клиента или на сервере, а JPython компилирует сценарии Python в код виртуальной машины Java, благодаря чему они могут выполняться в системах, поддерживающих Java, и беспрепятственно интегрировать библиотеки классов Java для использования в коде Python. В качестве инструментального средства, упрощающего создание сайтов, внимание вебмастеров и программистов CGI также привлек основанный на Python сервер веб-приложений Zope, описываемый в данной книге.

Поддержка

Что касается практической стороны, то коммерческая поддержка, консультации, готовые дистрибутивы и профессиональное обучение в настоящее время предоставляются многими фирмами. Например, интерпретатор Python можно получить на CD и в пакетах, продаваемых различными компаниями (в том числе Walnut Creek, «Dr. Dobb's Journal» и ActiveState); кроме того, Python обычно бесплатно поставляется в собранном виде во многих дистрибутивах операционной системы Linux.

Работа

Сейчас можно зарабатывать деньги в качестве Python-программиста (даже не прибегая к необходимости писать большие книги, содержащие интересные идеи). Когда писалась эта книга, на доске объявлений о работе на *http://www.python.org/Jobs.html* было перечислено около 60 компаний, которым требуются программисты Python в США или за рубежом. Поиск «Python» на популярных сайтах по трудоустройству дает еще более внушительные результаты: 285 связанных с Python рабочих мест на Monster.com и 369 на dice.com. Конечно, не обязательно менять работу, но приятно знать, что теперь можно зарабатывать на жизнь благодаря знанию языка, использование которого к тому же доставляет удовольствие.

Инструменты разработки

Python лег в основу многих проектов разработки инструментальных средств. Самыми заметными из них на момент написания книги являются проект Software Carpentry, в котором разрабатываются новые базовые программные инструментальные средства для Python; ActiveState, которая близка к выпуску продуктов разработки на Python для Windows и Linux[1]; и PythonWare, собирающаяся выпустить для Python интегрированную среду разработки и средство создания графического интерфейса пользователя.

Компиляторы

Когда писалось это предисловие, ActiveState объявила о создании нового компилятора Python для структуры Microsoft.NET и языковой среды C# – действительного компилятора Python и независимой реализации языка Python, в которой генерируются файлы DLL и EXE, можно разрабатывать код Python в Visual Studio и обеспечивается гладкая интеграция .NET со сценариями Python. Это будет третья реализация Python, наряду со стандартным Python, основанным на C, и системой Jpython, основанной на Java.

---

[1] ActiveState выпустила IDE Komodo, поддерживающий Python, для платформ Windows и Linux. – *Примеч. науч. ред.*

Так что же такое Python?

Если вам нужно исчерпывающее определение темы данной книги, воспользуйтесь таким:

Python является языком программирования общего назначения с открытым исходным кодом (open source), оптимизированным для качества, производительности, переносимости и интеграции. Им пользуются сотни тысяч разработчиков по всему миру в таких областях, как создание интернет-сценариев, системное программирование, проектирование пользовательских интерфейсов, настройка программных продуктов под пользователя и др.

Помимо всего остального Python поддерживает объектно-ориентированное программирование (ООП); обладает синтаксисом, который очень прост, легко читается и сопровождается; интегрируется с компонентами, написанными на языке C; обладает большим собранием уже запрограммированных интерфейсов и утилит. Несмотря на свое общее назначение, Python часто называют языком сценариев (*scripting language*), поскольку в нем просто использовать другие программные компоненты и управлять ими. Возможно, самым большим достоинством Python является просто то, что с его помощью разработка программного обеспечения становится более быстрой и приятной. Как это происходит, станет ясно из дальнейшего изложения.

Обучение

Python начал также привлекать внимание образовательных учреждений, многие из которых рассматривают его как «Pascal 2000-х годов» – язык, идеальный для обучения программированию благодаря его простоте и структуре. Отчасти это направление порождено проектом Гвидо ван Россума «Computer Programming for Everybody» (CP4E, компьютерное программирование для всех), который имеет целью сделать Python предпочтительным языком для начинающих программистов во всем мире. Будущее проекта CP4E пока остается неясным, но сформировалась группа особого интереса (SIG) к Python, которая должна заняться вопросами, связанными с образованием. Независимо от исхода той или иной инициативы, Python может сделать программирование более доступным для тех многочисленных людей, которым скоро наскучит щелкать по заранее запрограммированным ссылкам по мере их превращения из пользователей компьютеров в создателей сценариев.

Иными словами, 1995 год давно прошел. Большая часть приведенного перечня была невыполнима, когда было задумано первое издание этой книги. Естественно, этот список окажется устаревшим даже раньше, чем эта книга попадет на полки, но тем не менее, он показывает те вехи, которыми отмечено развитие Python за последние пять лет и которые ожидают нас в ближайшие годы. В качестве языка, оптимально подходящего к сегодняшним требованиям при создании программного обеспечения, Python, несомненно, еще ждет вершины своего успеха.

Для чего потребовалось это издание?

Одним из следствий растущей популярности Python стал наплыв новых пользователей, стилей программирования и приложений, из-за чего некоторые части первого издания данной книги потребовали обновления. Сам Python изменился незначительно, но важные расширения упростили различные стороны разработки на Python и заслуживают того, чтобы о них было рассказано.

Возможно, главной причиной этого издания является то, что изменилась «аудитория» Python. За последние пять лет Python превратился из развивающегося языка, которым интересуются преимущественно первопроходцы, в широко используемый программистами инструмент для решения повседневных задач. Данное издание переориентировано на эту новую аудиторию Python. Вы увидите, что теперь это более посвященная техническим деталям книга, в меньшей степени нацеленная на знакомство с языком и его популяризацию, чем на рассказ о том, как применять его к задачам программирования реального масштаба.

Объем изменений таков, что данное издание отчасти представляет совершенно новую книгу. Хочу выразить признательность тем читателям, которым понравилась первая книга; надеюсь, что тот же дух они обнаружат во втором издании. И хотя книга в значительной мере переработана, я попытался сохранить возможно большую часть материала и стиля первоначальной книги (в особенности шутки :-).

После выхода пять лет назад первого издания мне предоставилась возможность преподавать Python в США и за рубежом, и некоторые новые примеры отражают опыт, полученный в результате этого учебного процесса. Новые примеры областей применения отражают часто возникающие вопросы и интересы – как мои собственные, так и моих учеников. Преподавание Python практическим работникам, многие из которых теперь *вынуждены* использовать Python в своей работе, обусловило новый уровень связи с практикой, что вы заметите в выборе примеров и тем данного издания.

Другие примеры появились на свет как результат удовольствия, получаемого мной при программировании на Python. Да, удовольствия: я твердо уверен, что одним из самых больших неосязаемых достоинств Python является его способность вызывать радость программирования у новичков и снова вызывать эту радость у тех, кто годами трудился с использованием более требовательных инструментов. В данном издании будет продемонстрировано, что Python чрезвычайно облегчает применение более сложных, но полезных инструментов, таких как потоки, сокеты, GUI, веб-сайты и ООП – областей, которые могут показаться и утомительными и пугающими в традиционных компилируемых языках, таких как C и C++.

Честно говоря, даже после восьми лет пребывания в качестве добросовестного «питониста» (*Pythonista*)[1] мне все еще приятно заниматься программированием, когда я делаю это на Python. Python является чрезвычайно продуктивным языком, а рассмотрение его приложений доставляет в первую очередь эстетическое удовольствие. Надеюсь, что данное издание, так же как и предыдущее, продемонстрирует, как использовать преимущества Python в продуктивности, и отчасти передаст удовлетворение и восхищение, доставляемые таким средством быстрой разработки приложений, как Python.

Основные изменения, внесенные во второе издание

Конечно, лучше всего составить впечатление о какой-либо книге, прочтя ее. Но для тех, кто читал первое издание, в последующих нескольких разделах специально более подробно описывается, что нового внесено в данное издание.

---

[1] После выхода первого издания приверженцы Python стремились найти для себя название. Чаще всего встречается *Pythonista*, но горстка ветеранов упорно называют себя *Pythoneer* или еще как-нибудь. Pythonista имеет псевдовоинственный оттенок, что не обязательно отражает истинное лицо политики языка сценариев.

Издание обновлено в соответствии с Python 2.0

Данное издание обновлено в соответствии с Python 2.0, а материал по графическим интерфейсам пользователя (GUI) обновлен для Tk версий 8.0 и более поздних. Формально это обновление началось с Python 1.5.2, но все примеры перед публикацией были проверены под версией 2.0.

Для тех, кто любит мелочи: выпуск 2.0 был первым выпуском Python после перехода Гвидо в BeOpen, а 1.6 был последним выпуском у предыдущего работодателя Гвидо, CNRI. Перед самым завершением мной окончательного наброска книги и после выхода версии 2.0 Гвидо и основная команда разработчиков Python перешли из BeOpen в Digital Creations, родину Zope – инструментального набора создания веб-приложений, но это перемещение не зависит от выпусков Python (дополнительные подробности смотрите в главе 1 «Введение в Python»).

В версии 2.0 появились новые расширения языка, но 2.0 и 1.6 по содержанию сходны, и обновление только добавляет несколько функций. Примечательно, что большинство примеров из первого издания по-прежнему действует пять лет спустя с последними выпусками Python; для тех, которые не работали, потребовались незначительные исправления (например, формат вызова GUI и интерфейсы C API).

С другой стороны, хотя основы языка не сильно изменились с момента первого издания, в него был добавлен ряд новых конструкций, и мы их всех применим тут. В число этих новых характеристик Python входят пакеты модулей, классы исключений, псевдо-приватные аттрибуты классов, строки в Unicode, новый модуль регулярных выражений, новые функции Tkinter, такие как построитель таблиц, стандартные диалоги и меню верхнего уровня и т. д. В новом приложении приводится сводка всех важнейших изменений в Python между первым и вторым изданиями этой книги.

Помимо изменений в языке в данной книге представлены новые инструменты и приложения Python, появившиеся за последние годы. В их число входят интерфейс программирования IDLE, компилятор JPython (известный также как «Jython»), расширения Active Scripting и COM, структура создания веб-приложений Zope, серверные страницы Python (Python Server Pages – PSP), режим ограниченного исполнения, генераторы кода HTMLgen и SWIG, поддержка потоков, модули CGI и протоколов Интернета и многое другое (эти пять лет были активными). Такие приложения составляют самую суть второго издания.

Книга переориентирована на более подготовленную аудиторию

В данном издании программирование на Python представлено *усложненными примерами*. Чтобы стать профессионалом в Python, нужно решить две различные задачи: овладеть основами самого языка, а затем научиться применять его в приложениях. Решение второй (и более крупной) задачи в данном издании осуществляется путем представления библиотек Python, инструментальных средств и технологий программирования. Поскольку это совершенно иная задача, следует сказать здесь несколько слов о том, почему она была поставлена.

Поскольку во время появления первого издания этой книги других изданий по Python на горизонте заметно не было, оно было обращено сразу к очень широкой аудитории: и к новичкам и к гуру одновременно. С тех пор появилась другая книга O'Reilly, «Learning Python» («Изучаем Python»), которая была ориентирована на новичков, и был опубликован «Python Pocket Reference» («Карманный справочник по Python») для читателей, нуждающихся в кратком справочнике по Python. В результате ввод-

ный материал по основам языка и первоначальные справочные приложения были изъяты из этой книги.

«Изучаем Python» знакомит с основами языка – синтаксисом, типами данных и т. д. – с помощью намеренно упрощенных примеров. Многие считают эту книгу идеальной для изучения самого языка, но Python может стать еще интереснее, когда вы овладеете базовым синтаксисом и сможете писать простые примеры в интерактивном интерфейсе. Научившись разрезать список, вы весьма скоро захотите делать реальные вещи, например писать сценарии для сравнения каталогов файлов, отвечать на запросы пользователя через Интернет, выводить графические изображения в окнах, читать электронную почту и т. д. Повседневная работа состоит в основном в применении языка, а не в самом языке.

Данная книга «Программирование на Python» сконцентрирована на «всем остальном» в разработках на Python. В ней рассказывается о библиотеках и инструментарии, находящихся за рамками основного языка, которые приобретают первостепенное значение при написании реальных приложений. Она также рассматривает вопросы проектирования более крупных приложений, например повторное использование кода и ООП, которые можно проиллюстрировать только в контексте программ более реального масштаба. Иными словами, книга «Программирование на Python», особенно в этом новом издании, призвана подхватить изложение там, где оно закончилось в «Изучаем Python».

Поэтому, если данная книга покажется вам слишком сложной, я советую предварительно прочесть «Изучаем Python» и после овладения основами вернуться сюда за продолжением рассказа. Если только у вас нет значительного опыта в программировании, это издание лучше всего использовать в качестве второй своей книги по Python.

В книге раскрыты новые темы

Большинство изменений в данном издании было сделано для того, чтобы осветить новые темы. В нем появились новые главы и разделы, рассказывающие о сценариях в Интернете, сценариях CGI, интерфейсах операционных систем, генераторе интегрирующего кода SWIG, более сложных разделах Tkinter, генераторе веб-страниц HTMLgen, JPython, потоках, режиме ограниченного исполнения и многом другом. Весь объем можно уяснить, обратившись к оглавлению, а здесь перечислены некоторые из новых тем и структурных изменений, которые появились в данном издании:

Темы

Уделено значительно большее внимание Интернету, системному программированию, графическим интерфейсам Tkinter и интеграции с C. Можно утверждать, что на них теперь в основном сосредоточен данный текст. Например, появилось шесть новых глав по сценариям в Интернете, в которых рассказано об инструментах на стороне клиента, сценариях на стороне сервера, веб-сайтах и более сложных темах и системах, связанных с Интернетом. Четыре новые главы посвящены системным вопросам: потокам, обработке каталогов, запуску программ и т. д. Материал по GUI также вырос с одной главы до значительно более полного представления в четырех главах, охватывая теперь все графические элементы (в том числе текстовые и холст), а также новую поддержку таблиц, меню и диалогов.

Интеграция с C

Главы, посвященные расширениям и встраиванию C, дополнены новым материалом, охватывающим такие темы, как SWIG (способ, которым в настоящее время следует соединять Python с библиотеками C/C++), и представляющим новые примеры смешанного режима, такие как диспетчер функций обратного вызова (рас-

ширение и встраивание). Интеграция с C лежит в сердцевине многих систем Python, но примеры из этой области неизбежно сложны и включают в себя большие программы на C, интересные только пользователям C. Из уважения к читателям, которым не требуется интегрировать Python с C, этот материал теперь помещен отдельно в конце книги. Некоторые листинги кода на C также удалены, чтобы сократить объем книги: вместо них я предпочитал по возможности отсылать читателей к исходным файлам на C на прилагаемом CD.

Хотя более поздние главы основываются на материале более ранних глав, темы в данном издании освещаются достаточно независимо и объединяются в части книги. Благодаря этому не будет большой натяжкой считать, что в данном издании объединены в одно целое четыре или пять книг. Структура основных разделов книги подчеркивает ее концентрацию на темах приложения:

Два замечания. Не дайте заглавиям ввести вас в заблуждение: хотя большинство разделов относится к темам приложений, все же попутно рассматриваются характеристики и общие идеи проектирования языка Python в контексте практических задач. И второе: читатели, использующие Python в качестве самостоятельного инструмента, могут благополучно пропустить главы, связанные с интеграцией, хотя все же рекомендуется бегло взглянуть на них. Программирование на C не дает такого удовольствия и легкости, как программирование на Python. Однако поскольку интеграция играет центральную роль в использовании Python в качестве языка сценариев, некоторое понимание ее может оказаться полезным независимо от того, занимаетесь ли вы интеграцией, написанием сценариев или тем и другим вместе.

Читатели первого издания обратят внимание на то, что материал по большей части является новым, причем даже главы, сохранившие свое название, сильно обновились. Заметно отсутствие в данном издании первоначального краткого обзора, минисправочника и учебного приложения, а также полное отсутствие прежней части II, что отражает новую направленность книги и предполагаемую аудиторию.

Книга более ориентирована на примеры

Эта книга в значительной мере состоит из примеров. В настоящем издании старые примеры расширены, чтобы придать им большую реалистичность (например, PyForm и PyCalc); кроме того, всюду добавлены новые примеры. В число основных примеров входят:

PyEdit

Объект и программа редактора текстовых файлов на Python/Tk.

PyView

Слайд-шоу для фотоизображений и файлов заметок.

PyDraw

Графический редактор для рисования и перемещения объектов изображений.

PyTree

Программа для изображения древовидных структур данных.

PyClock

Графический элемент для изображения аналоговых и цифровых часов на Python/Tk.

PyToe

Графическая программа игры в крестики-нолики с искусственным интеллектом.

PyForm

Броузер для таблиц постоянных объектов.

PyCalc

Графический элемент калькулятора на Python/Tk.

PyMail

Почтовый клиент с поддержкой POP и SMTP на Python/Tk.

PyFtp

Простой GUI для пересылки файлов на Python/Tk.

PyErrata

Размещаемая в Сети система сообщений об ошибках.

PyMailCgi

Размещаемый в Сети интерфейс электронной почты.

Есть также новые примеры интеграции с C в смешанном режиме (например, регистрация функции обратного вызова (callback) и работа с объектом класса), примеры SWIG (с «теневыми» классами для C++ и без них), дополнительные примеры Интернета (сценарии отправки и получения по FTP, примеры NNTP и HTTP, средства работы с электронной почтой и новые примеры модулей `socket` и `select`), много новых примеров потоков в Python и новое освещение JPython, HTMLgen, Zope, Active Scripting, COM и интерфейсов Python к базам данных. Многие новые примеры являются довольно продвинутыми, впрочем, как и весь текст.

Кроме того, прежний API для встраивания в C кода на Python (называемый теперь *ppembed*) расширен с целью поддержки предварительной компиляции строк в байт-коды, а первоначальный пример калькулятора (имеющий теперь название *PyCalc*) усилен и стал поддерживать ввод с клавиатуры, буфер команд, цвета и другие функции.

На практике примеры в новой книге, распространяемые на прилагаемом к данному изданию CD-ROM, *сами по себе* являются довольно сложной системой программ на Python, входящие в которую примеры структурно изменены в ряде важных направлений:

Дерево примеров

Весь дистрибутив примеров организован в виде одного большого *пакета модулей* Python, что облегчает импорт из других каталогов и устраняет конфликт имен с другим кодом Python, установленным на компьютере. Использование путей каталогов в командах import (вместо сложного PYTHONPATH) облегчает также установление происхождения модулей. Более того, теперь нужно добавить в путь поиска PYTHONPATH только один каталог для всего дерева примеров в книге: каталог, содержащий корневой каталог примеров *PP2E*. Для использования примеров, приведенных в книге, в собственных приложениях нужно просто импортировать их через корень пакета *PP2E* (например, `from PP2E.Launcher import which`).

Имена файлов примеров

Имена модулей стали значительно менее таинственными. Я давно перестал обращать внимание на совместимость с форматом DOS 8.3 и использую более описательные имена. Были также исправлены некоторые старые имена файлов, полностью записанные в верхнем регистре – последнее наследие MS-DOS.

Названия примеров

В метках листингов примеров теперь указывается полный путь к файлу исходного кода примера, что помогает найти его в дистрибутиве примеров. Например, файл с исходным кодом примера, имя которого указано как *Example N-M: PP2E\Internet\Ftp\sousa.py*, указывает на файл *sousa.py* в подкаталоге *PP2E\Internet\Ftp* каталога дистрибутива примеров.[1]

Командные строки примеров

Аналогично, когда показана командная строка, введенная после приглашения, например `C:\...\PP2E\System\Streams>`, она в действительности должна быть введе-

[1] «Каталог дистрибутива примеров» является каталогом, содержащим каталог верхнего уровня *PP2E* дерева примеров книги. На CD это самый верхний каталог *Examples*; если вы скопировали примеры на диск, это то место, куда вы скопировали (или распаковали) корневой каталог *PP2E*. Большинство этих примеров можно непосредственно запускать с CD, но они должны быть скопированы на жесткий диск для внесения в них изменений и чтобы позволить Python сохранять файлы компилированного байт-кода *.pyc*, допускающие более быстрый начальный запуск.

на так, чтобы указывать на подкаталог *PP2E\System\Streams* в вашем дереве примеров. Пользователям Unix и Linux: пожалуйста, имейте в виду / при встрече с \ в путях к файлам (официальное извинение по этому поводу выражено в следующем разделе).

Средства запуска примеров

Поскольку приятно иметь возможность сразу щелкнуть куда нужно мышкой, есть новые самоконфигурирующиеся программы запуска демонстрационных примеров (описываемые ниже в этом предисловии в разделе «Вкратце»), позволяющие быстро взглянуть на сценарии Python в действии с минимальной предварительной настройкой. Обычно можно запускать их прямо с CD, не устанавливая каких-либо переменных окружения.

Книга более нейтральна в отношении используемых платформ

За исключением все тех же примеров с интеграцией с C, большинство программ в этом издании было разработано на моем портативном компьютере под Windows 98 с прицелом на переносимость под Linux и другие платформы. В действительности некоторые примеры порождены моим желанием создать на Python переносимые эквиваленты некоторых инструментов, отсутствующих в Windows (например, программ для разбивки файлов на части). Когда программы показываются в действии, это обычно делается на Windows; на платформе Red Hat Linux 6.x они демонстрируются только при использовании специфических для Unix интерфейсов.

Это вовсе не политическое заявление: Linux мне тоже нравится. Это в основном следствие того, что я писал эту книгу с помощью MS Word; когда время ограничено, удобнее запускать сценарии на той же платформе, что и издательские средства, чтобы не перегружаться слишком часто в Linux. К счастью, поскольку Python стал теперь в большой степени переносим между Windows и Linux, используемая операционная система меньше заботит разработчиков Python, чем было ранее. Python, его библиотеки и система GUI Tkinter в настоящее время прекрасно работают на обеих платформах.

Однако поскольку я не занимаюсь политикой, то постарался сделать примеры возможно более независимыми от платформы и попутно указать на специфические для каждой платформы проблемы. Вообще говоря, большинство сценариев должно работать на обычных платформах Python без внесения изменений. Например, все примеры GUI были проверены под Windows (98, 95) и Linux (KDE, Gnome), а большинство примеров командной строки и потоков были разработаны в Windows, но работают и в Linux. Поскольку системные интерфейсы Python обычно строятся в расчете на переносимость, сделать это легче, чем может показаться.

В то же время, эта книга при необходимости погружается в специфические для платформ темы. Заново изложены многие специальные темы Windows: Active Scripting, COM, варианты запуска программ и т. д. Читатели, работающие под Linux и Unix, найдут и материал, относящийся к их платформе: ветвление, конвейеры и тому подобное. Заново изложены также способы редактирования и запуска программ Python на большинстве главных платформ.

Единственное место, где читатели смогут усмотреть уклон в сторону определенной платформы, – это примеры интеграции Python с C. Для простоты детали компиляции программ на C, о которых говорится в данной книге, все еще имеют некоторый уклон в сторону Unix/Linux. Во всяком случае этому можно найти разумное объяснение: Linux не только поставляется с бесплатными компиляторами C, но вокруг этого языка выросла вся его среда разработки. Код расширений на C, приведенных в этой книге,

Но все равно это не справочное руководство

Обратите, пожалуйста, внимание на то, что это издание, как и первое, все же скорее *учебник*, чем справочное руководство (несмотря на название, сходное с популярным справочником по Perl). Эта книга должна учить, а не документировать. С помощью ее оглавления и предметного указателя можно найти конкретные детали, и новая структура облегчает это. Но все же данное издание задумано как книга, которую используют вместе со справочниками по Python, а не вместо них. Поскольку руководства по Python бесплатны, написаны хорошо, доступны в Сети и часто редактируются, было бы безумием тратить место на копирование их содержания. Исчерпывающий список всех инструментов, имеющихся в системе Python, можно найти в других книгах (например, изданной O'Reilly «Python Pocket Reference») или стандартных руководствах на веб-сайте Python либо на компакт-диске, прилагаемом к данной книге.

будет работать и под Windows, но может потребоваться использование различных процедур сборки программ в зависимости от используемого в Windows компилятора. Издательство O'Reilly опубликовало замечательную книгу «Python Programming on Win32», в которой освещены специфические для Windows темы Python вроде этой, которая должна помочь справиться с некоторыми наблюдаемыми здесь расхождениями. Если вы занимаетесь программированием под Windows, то найдете все относящиеся к Windows подробности, которые здесь пропущены.

Использование примеров и демонстрационных программ

Хочу здесь кратко описать, как использовать имеющиеся в книге примеры. В целом, однако, посмотрите, пожалуйста, следующие текстовые файлы в каталоге дистрибутива примеров, в которых вы найдете дополнительные сведения:

- *README-root.txt* – замечания о структуре пакета
- *PP2E\README-PP2E.txt* – общие замечания об использовании
- *PP2E\Config\setup-pp.bat* – конфигурация для Windows
- *PP2E\Config\setup-pp.csh* – конфигурация для Unix и Linux

Из этих файлов наиболее информативен *README-PP2E.txt*, а в каталоге *PP2E\Config* содержатся все файлы примеров конфигурации. Здесь дается обзор, но в перечисленных файлах находится полное описание.

Вкратце

Если вы сразу хотите посмотреть некоторые примеры Python, поступите так:

1. Установите Python с компакт-диска, прилагаемого к книге, если он еще не установлен на вашем компьютере. В Windows щелкните на имени самоустанавливающейся программы на CD и сделайте установку по умолчанию (отвечайте «yes» или «next» на все приглашения). Для других систем посмотрите файл README (заархивированный дистрибутив исходного кода на CD можно использовать для локальной сборки Python).
2. Запустите один из следующих *самоконфигурирующихся сценариев*, располагающихся в каталоге CD верхнего уровня Examples\PP2E. Щелкните по соответству-

ющим пиктограммам в проводнике или запустите из системной подсказки (например, окна консоли DOS или Linux Xterm) посредством командной строки формата `python имя-сценария` (может потребоваться указать полный путь к `python`, если он не включен в систему):

- *Launch_PyDemos.pyw* – главная инструментальная панель для запуска демонстрационных программ Python/Tk
- *Launch_PyGadgets_bar.pyw* – инструментальная панель для запуска утилит Python/Tk
- *Launch_PyGadgets.py* – запускает стандартные утилиты Python/Tk
- *LaunchBrowser.py* – открывает указатель веб-примеров в веб-броузере

Сценарии `Launch_*` запускают программы Python переносимым образом[1] и требуют только наличия установленного Python: для их запуска не требуется предварительной настройки переменных окружения или изменений в прилагаемых файлах настройки *PP2E\Config*. `LaunchBrowser` сможет работать, если найдет на вашей машине веб-броузер, даже если у вас нет соединения с Интернетом (хотя некоторые интернет-примеры работают неполным образом в отсутствие работающего соединения).

Если отказаться от установки Python, все же можно запустить несколько веб-демонстраций Python, отправив броузер на *http://starship.python.net/~lutz/PyInternetDemos.html*. Так как эти примеры предназначены для выполнения сценариев на сервере, лучше всего они работают при запуске на этом сайте, а не с CD, прилагаемого к книге.

Подробности

Для лучшей организации новых примеров я поместил в каталог верхнего уровня дистрибутива примеров *PP2E* программу запуска демонстраций, *PyDemos.pyw*. На рис. П.1 показана PyDemos в действии под Windows после нажатия нескольких кнопок. Панель запуска появляется в левой части экрана; с ее помощью можно щелчком мыши запустить большинство графических примеров, приведенных в книге. Если на вашей машине удастся обнаружить броузер, то с помощью панели запуска демонстраций можно также запускать основные примеры для Интернета (смотрите описание программы запуска далее).

Помимо запуска демонстрационных программ исходный код PyDemos дает указатели на основные примеры в дистрибутиве; детали ищите в исходном коде программы. В каталоге верхнего уровня примеров вы также обнаружите сценарии для автоматической компиляции под Linux примеров интеграции Python и C, которые служат указателями для основных примеров на C.

Я также включил программу верхнего уровня под названием *PyGadgets.py* и родственную ей *PyGadgets_bar.pyw*, чтобы запускать для реального использования, а не для демонстрации некоторые наиболее полезные примеры GUI из книги (в большинстве своем часто используемые мной программы; переконфигурируйте их, если захотите). На рис. П.2 показано, как выглядит в Windows `PyGadgets_bar`, а также некоторые утилиты, которые могут запускаться его кнопками. Все эти программы представлены в данной книге и включены в дистрибутив примеров. Для большинства из

[1] Все демо- и запускающие сценарии написаны переносимым образом, но на момент написания книги известно, что они работают только под Windows 95/98 и Linux; для других платформ могут потребоваться незначительные изменения. Приношу извинения, если вы используете платформу, которую я не смог проверить: Tk работает под Windows, X11 и на Macs; сам Python работает всюду – от карманных PDA до мэйнфреймов; и мой вклад в написание этой книги был не столь велик, как вы могли подумать.

Рис. П.1. Запускающая программа PyDemos со всплывающими окнами и демонстрационными программами (фотография Гвидо перепечатана с разрешения Dr. Dobb's Journal)

них требуется Python с поддержкой Tkinter, но это и есть стандартная конфигурация для запуска материалов с CD этой книги под Windows.

Для непосредственного запуска файлов, перечисленных в предыдущем параграфе, требуется настроить путь поиска модулей Python (подсказку ищите в файлах `PP2E/Config/setup*`). Но если вы хотите запустить собрание демонстрационных программ Python из данной книги и не хотите утруждать себя предварительной настройкой окружения, просто выполните сценарии утилит в каталоге *PP2E*: *Launch_PyDemos.pyw*, *Launch_PyGadgets.py* и *Launch_ PyGadgets_bar.pyw*.

Эти сценарии для запуска программ, написанные на Python, предполагают, что Python уже установлен, и автоматически найдут интерпретатор языка и дистрибутив примеров из книги, а также настроят пути к модулям Python и системные пути поиска так, как это требуется для запуска демонстрационных программ. Вероятно, вы сможете запустить эти сценарии, просто щелкая по их именам в проводнике, а также сможете запустить их непосредственно с прилагаемого CD-ROM. Дополнительные сведения можно найти в комментариях в начале *Launcher.py* (или прочесть об этих сценариях в главе 4 «Более крупные системные примеры, часть 1»).

Многие из примеров для Интернета, использующих броузеры, можно найти в сети на *http://starship.python.net/~lutz/PyInternetDemos.html*, где можно проверить их работу. Поскольку эти примеры выполняются в броузере, их можно посмотреть, даже если на машине не установлен Python (или поддержка Tk для Python).

Программа PyDemos также пытается запустить броузер с веб-страницами основных примеров путем выполнения сценария *LaunchBrowser.py* в корневом каталоге примеров. Этот сценарий обычно успешно находит на вашей машине броузер, которым можно воспользоваться; если поиск окажется неудачным, смотрите дополнительные

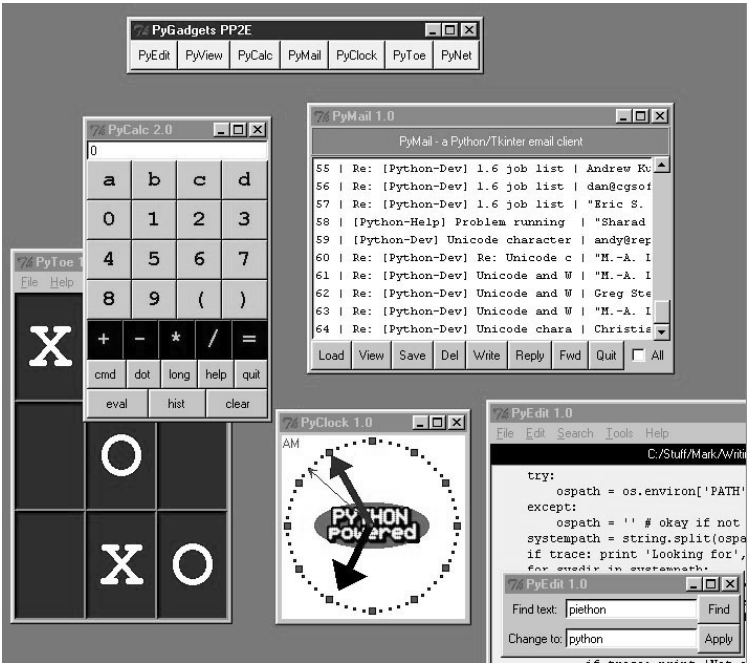


Рис. П.2. Панель запуска утилит PyGadgets вместе с программами

сведения в сценарии. Если `LaunchBrowser` сможет найти на вашей машине броузер, некоторые кнопки автоматически выведут веб-страницы независимо от наличия действующего соединения с Интернетом (если соединения нет, в броузере будут показаны локальные файлы). На рис. П.3 показано, как выглядит страница PyInternetDemos в Internet Explorer под Windows.

Особенно интересно, что ссылка *getfile.html* на этой странице позволяет просмотреть исходный код любого другого файла на сайте книги – код HTML, CGI-сценарии Python и т. д.; подробности смотрите в главе 12. Подведем итоги; вот что вы найдете в каталоге верхнего уровня *PP2E* дистрибутива примеров книги:

PyDemos.pyw

Панель кнопок для запуска основных примеров GUI и Интернета.

PyGadgets.py

Запуск программ в недемонстрационном режиме для обычного использования.

PyGadgets_bar.pyw

Панель кнопок для запуска PyGadgets по требованию.

*Launch_*.py**

Запускает программы PyDemos и PyGadgets с помощью *Launcher.py*, автоматически конфигурирующего пути поиска (запускайте их, чтобы бегло взглянуть).

Launcher.py

Используется для запуска программ без настройки окружения: находит Python, устанавливает PYTHONPATH, запускает программы Python.

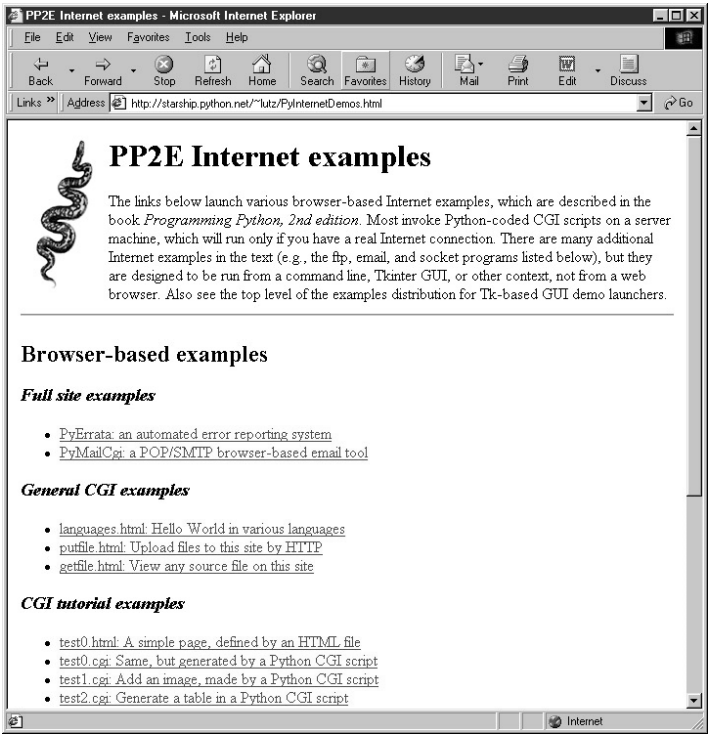


Рис. П.3. Веб-страница PyInternetDemos

LaunchBrowser.py

Открывает веб-страницы примеров в автоматически найденном броузере, загружая их из Сети или из локальных файлов; при непосредственном запуске открывает страницу указателя PyInternetDemos.

Существуют также подкаталоги для примеров из каждой основной группы тем, рассматриваемых в книге.

Кроме того, каталог верхнего уровня *PP2E\PyTools* содержит написанные на Python утилиты командной строки для преобразования символов перевода строки во всех текстовых файлах примеров в формат DOS или Unix (полезно использовать, если они странно выглядят в вашем текстовом редакторе), превращения всех файлов примеров в доступные для записи (полезны, если вы перетаскиваете файлы с прилагаемого CD), удаления старых файлов байт-кода *.pyc* в дереве каталогов и другие. Дополнительные сведения, касающиеся всех вопросов, связанных с примерами, ищите в файле *README-PP2E.txt*.

Где все это находится

Дистрибутив примеров книги находится на сопровождающем ее компакт-диске. Детали использования описаны в файле README на верхнем уровне каталогов CD. Для беглого ознакомления можно просмотреть корневой каталог примеров на компакт-диске в своем любимом файловом проводнике.

Помимо примеров из книги CD содержит также различные относящиеся к Python пакеты, в том числе полную *программу самоустановки* под Windows с поддержкой Python и Tk (для установки дважды щелкните по ней и ответьте «yes» на все приглашения), полный *дистрибутив исходного кода* Python (распакуйте и откомпилируйте на своей машине) и набор *стандартной документации* Python в формате HTML (щелкните для просмотра в своем веб-броузере).

Дополнительные пакеты открытого исходного кода, например последние версии (на момент публикации) генератора кода SWIG и Jpython, тоже помещены на CD, но вы всегда можете найти новейшие версии Python и других пакетов на веб-сайте Python *http://www.python.org*.

Использованные в книге типографские обозначения

В данной книге принято следующее использование шрифтов:

Курсив

Используется в именах файлов и каталогов, URL, командах, для выделения новых терминов при первом знакомстве и в некоторых комментариях в фрагментах кода.

`Моноширинный`

Используется в листингах кода и для обозначения модулей, методов, опций, классов, функций, утверждений, программ, объектов и тегов HTML.

`Моноширинный полужирный`

Используется в приводимом коде для показа данных, вводимых пользователем.

`Моноширинный курсив`

Используется для обозначения заменяемого пользователем текста.

Изображение совы обозначает замечание, относящееся к близлежащему тексту.

Изображение индюшки обозначает предупреждение, относящееся к близлежащему тексту.

Где можно найти обновления

Как и прежде, обновления, исправления и дополнения для данной книги поддерживаются на веб-сайте автора *http://www.rmi.net/~lutz*. Найдите на этой странице ссылку на второе издание, чтобы получить всю дополнительную информацию, касающуюся данной версии книги. Как и для первого издания, я буду вести на этом сайте журнал регистрации изменений, производимых в Python, который следует рассматривать как дополнение к данной книге.

Начиная с этого издания я открываю в Сети систему сообщения пользователями об ошибках, найденных в книге, на сайте:

http://starship.python.net/~lutz/PyErrata/pyerrata.html

Там вы найдете формы для сообщений о проблемах, возникших с книгой, и комментариев, а также сможете просматривать базу данных сообщений, отсортированную по различным ключам. По умолчанию сообщения хранятся в базе данных общего доступа, но при желании можно направлять их закрытым письмом. Система PyErrata тоже написана на Python и составила пример, вошедший в данную книгу; смотрите главу 14. Рис. П.4 показывает внешний вид корневой страницы PyErrata.

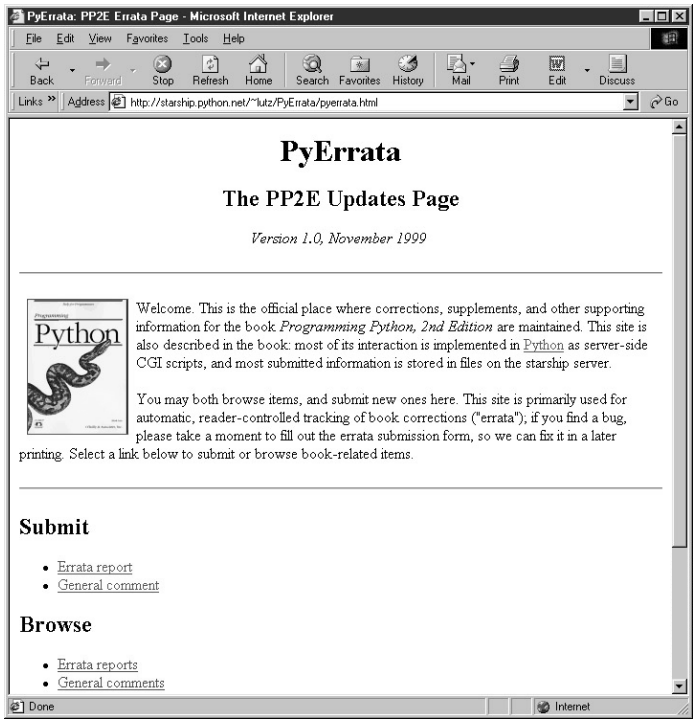


Рис. П.4. Сайт PyErrata вносимых в книгу исправлений

Если какие-либо из этих адресов окажутся недействующими, то получить доступ к этим страницам можно также на веб-сайте издательства O'Reilly *http://www.oreilly.com*.[1] Я по-прежнему буду рад получать прямые письма читателей, но надеюсь, что PyErrata усовершенствует процесс передачи сообщений об ошибках.

Связь с O'Reilly

Можно также обратиться с комментариями и вопросами относительно данной книги к издателю:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)

[1] На веб-сайте O'Reilly тоже есть система приема сообщений об ошибках, и оба эти списка в совокупности следует рассматривать как официальное заключение о найденных ошибках и внесенных изменениях.

(707) 829-0515 (international/local)
(707) 829-0104 (fax)

В O'Reilly есть веб-страница, посвященная данной книге, на которой приведены ошибки, примеры и любая дополнительная информация. Эта страница находится на

http://www.oreilly.com/catalog/python2/

Для комментариев или технических вопросов по данной книге пишите по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, конференциях, программном обеспечении, центрах ресурсов и сети O'Reilly Network можно найти на веб-сайте O'Reilly на

http://www.oreilly.com

Благодарности

Помимо людей, упомянутых в первом издании, я хотел бы дополнительно выразить признательность тем, кто так или иначе помогал мне во время данного проекта второго издания:

- Редактору первого издания Фрэнку Уиллисону (Frank Willison) за то, что он просмотрел эту редакцию и продвигал Python как в O'Reilly, так и за его пределами. Следующему редактору данной книги Лоре Левин (Laura Lewin), перенявшей у него эстафету.
- Создателю Python Гвидо ван Россуму (Guido van Rossum), благодаря которому работа над книгой снова стала удовольствием.
- Участникам рецензирования раннего проекта данного издания: Эрику Рэймонду (Eric Raymond), Марку Хэммонду (Mark Hammond), Дэвиду Эшеру (David Ascher), Тиму Петерсу (Tim Peters) и Дэйву Бизли (Dave Beazley).
- Тиму О'Рейлли (Tim O'Reilly) и служащим O'Reilly & Associates как за выход этой книги, так и за поддержку программного обеспечения с открытым кодом в целом.
- Сообществу Python вообще за прилежание, напряженную работу и юмор – в прежние годы и сейчас. Мы далеко продвинулись, но вспомним строчку из 1970-х: «Вы еще ничего не видите».
- Студентам многочисленных курсов Python, которым я преподавал, а также многочисленным читателям, не пожалевшим времени, чтобы послать мне комментарии по поводу первого издания: ваше мнение помогло определить характер данной новой редакции.

Наконец, несколько личных выражений благодарности. Моим детям, Майклу, Саманте и Роксане, за целеустремленность. Если они представляют свое поколение, то будущее нашего вида находится в хороших руках. Вам придется извинить меня за гордыню, но с такими детьми, как мои, невозможно чувствовать иначе.

А самая большая благодарность Лайзе, матери этих изумительных детей. Я в самом большом долгу перед ней за все – от терпеливого отношения к моим уходам от реальности, когда я пишу книги вроде этой, до того, что она спасла меня от тюрьмы в годы нашей молодости. Что бы ни таило нам будущее, я благодарен судьбе, соединившей нас два десятилетия назад.

Марк Лутц
Ноябрь 2000
Где-то в Колорадо

«Когда Билли пойдет вниз, с ним это произойдет быстро»

За последние пять лет наблюдался также подъем движения за программное обеспечение с открытым кодом (open source) – это программы, бесплатно распространяемые вместе с полным исходным кодом, обычно являющиеся результатом работы многих разработчиков, сотрудничающих друг с другом в свободной манере. В эту категорию попадают Python, операционная система Linux и многие другие инструменты, например Perl и веб-сервер Apache. Частично благодаря вызову, брошенному им преобладанию на рынке мегакомпаний, движение программного обеспечения с открытым кодом быстро и глубоко проникло в общество.

Позвольте рассказать о событии, недавно подчеркнувшем степень влияния на меня этого движения. Чтобы вам стал понятен этот рассказ, следует сообщить, что я писал эту книгу в небольшом городке в Колорадо, о котором нельзя сказать, что он находился на переднем крае технологических нововведений. Сказать образнее, это одно из тех мест, которые называют «ковбойский городок».

Я зашел в небольшую местную книжную лавку в поисках последнего номера *Linux Journal*. После недолгих поисков я нашел требуемое и направился к кассе. За прилавком стояли двое служащих, внешний вид которых более соответствовал занятию родео, чем нахождению за прилавком этого заведения. Старший из них был седым, с усами и с задубелой кожей человека, привыкшего к жизни на ранчо. Оба носили обязательные бейсбольные кепки. Вылитые ковбои.

Когда я положил журнал, старший из служащих на некоторое время поднял глаза и сказал, типично по-ковбойски растягивая слова: «Угу, Линукс? Вот что я скажу: когда Билли пойдет вниз, с ним это произойдет быстро!» Разумеется, речь шла о широко комментировавшемся соревновании между Linux и Microsoft Windows Билла Гейтса, усилившемся благодаря движению open source.

В другое время эти двое могли бы рассуждать о коровах и револьверах, сидя за чашкой крепкого кофе. Однако каким-то странным образом они стали страстными защитниками Linux – операционной системы с открытым исходным кодом. Подобрав отвалившуюся челюсть, я вступил с ними в оживленный разговор о Linux, Microsoft, Python и всяких открытых вещах. Можно сказать, мы славно поболтали.

Я не хочу отдавать предпочтение одной операционной системе перед другой: у каждой из них есть свои преимущества, а Python одинаково хорошо работает на той и другой платформе (в действительности вошедшие в книгу примеры разрабатывались на обеих системах). Но меня поразило, что мысль, которую разработчики программного обеспечения часто считают само собой разумеющейся, имела такое глубокое влияние на средних американцев. Это вселяет в меня большие надежды: если технология действительно должна повысить качество жизни в следующем тысячелетии, нам нужно побольше таких ковбоев.

1

Знакомство с Python

«А теперь нечто совершенно новое»

Эта книга посвящена использованию Python, объектно-ориентированного языка программирования весьма высокого уровня с открытым исходным кодом[1], предназначенного для оптимизации скорости разработки. Несмотря на то что Python в полной мере является языком общего назначения, его часто называют объектно-ориентированным *языком сценариев* – отчасти просто из-за легкости использования, отчасти из-за частого применения при «оркестровке» или «склеивании» в приложении других программных компонентов.

Если вы новичок в Python, то, возможно, все же слышали где-нибудь об этом языке, но точно не знаете, в чем его особенности. В помощь начинающим эта глава неформально знакомит с характеристиками Python и той ролью, которую он играет. В этом знакомстве будет больше пользы, если познакомиться с реальными программами Python, но прежде чем заняться деревьями, взглянем на лес в целом.

В предисловии говорилось, что для Python характерны такие понятия, как качество, продуктивность, переносимость и интеграция. Поскольку в эти четыре слова сведено большинство причин использования Python, хотелось бы определить их более подробно:

Качество

С помощью Python легко писать программы, которые можно повторно использовать и сопровождать. Он специально спроектирован так, чтобы повысить стандарт качества разработок в области языков сценариев. Понятный синтаксис Python и ясная архитектура почти вынуждают программистов писать код, отличающийся хорошей читаемостью, что важно для программного обеспечения, которое могут изменять другие программисты. Язык Python действительно создает впечатление планомерности, а не накопления характеристик. Кроме того, Python обладает хорошими возможностями для поддержки современных методологий повторного использования программного обеспечения. В действительности создание высококачественных компонентов Python, которые могут применяться во многих контекстах, происходит почти автоматически.

Продуктивность

Python оптимизирован по скорости разработки программного обеспечения. На нем можно быстро писать программы, потому что детали, которые в языках более

---

[1] Системы с открытым исходным кодом иногда называются *freeware*, поскольку их исходный код распространяется бесплатно и контролируется сообществом. Однако пусть это понятие не вводит вас в заблуждение, так как при наличии на сегодняшний день в этом сообществе примерно полумиллиона пользователей Python имеет очень хорошую поддержку.

низкого уровня вы должны кодировать явно, берет на себя интерпретатор. В сценариях Python не найти таких вещей, как объявление типов, управление памятью или процедуры компиляции. Но быстрая первоначальная разработка – лишь одна составляющая продуктивности. В реальном мире программисты пишут код как для компьютера, который будет его выполнять, так и для других программистов, которые будут его читать и сопровождать. Поскольку синтаксис Python напоминает *исполняемый псевдокод*, создаваемые программы легко понять через длительное время после того, как они написаны. Кроме того, Python поддерживает (но не навязывает) продвинутые парадигмы, например объектно-ориентированное программирование, которое еще больше повышает производительность и сокращает сроки разработки.

Переносимость

Большинство программ Python без изменений работает почти на всех используемых на сегодняшний день компьютерных системах. Фактически программы Python сегодня работают везде – от мэйнфреймов IBM и суперкомпьютеров Cray до ноутбуков и карманных PDA. Хотя для некоторых платформ существуют непереносимые расширения, базовый язык Python и его библиотеки нейтральны в отношении платформы. Например, сценарии Python, разработанные на Linux, в большинстве своем могут сразу выполняться под Windows, и наоборот – просто скопируйте сценарий. Более того, программа с графическим интерфейсом пользователя (GUI), написанная со стандартной библиотекой Python Tkinter, будет выполняться в системе X Windows, Microsoft Windows и Macintosh, выглядя при этом в каждом случае как родная, без всякой модификации исходного кода.

Интеграция

Python спроектирован с возможностью интеграции с другими инструментами. Написанные на Python программы можно легко смешивать с другими компонентами системы и управлять ими. Например, сценарии Python могут сейчас обращаться к имеющимся библиотекам C и C++, работать с классами Java, интегрироваться с компонентами COM и CORBA и многое другое. Кроме того, программы, написанные на других языках, могут с такой же легкостью запускать сценарии Python, вызывая функции API C и Java, обращаясь к COM-серверам, написанным на Python, и т. д. Python – это не запертый ящик.

В эпоху неуклонно сжимающихся графиков разработки, более быстрых машин и гетерогенных приложений такие достоинства оказываются сильными союзниками как в малых, так и в больших проектах. Естественно, есть и другие особенности Python, привлекательные для разработчиков, например легкость изучения для разработчиков и пользователей, библиотеки готовых инструментов, минимизирующие объем предварительных разработок, и полная бесплатность, сокращающая расходы на разработку продукта и его развертывание.

Но самым привлекательным и определяющим качеством Python является, вероятно, его нацеленность на продуктивность. Когда я пишу эти строки, главная проблема, существующая в мире разработки программного обеспечения, состоит не просто в том, чтобы быстро писать программы, а в том, чтобы найти разработчиков, у которых вообще есть время для разработки программ. Время, затрачиваемое разработчиками, приобрело первостепенное значение, гораздо более важное, чем скорость выполнения. Существующих проектов просто больше, чем программистов, которыми можно их укомплектовать.

В качестве языка, оптимизированного для продуктивности разработчиков, Python дает правильный ответ на вопросы, возникающие в мире разработки программного обеспечения. Программируя на Python, разработчики не только могут быстро реали-

зовывать системы, но получаемые системы легче сопровождать, они переносимы и просто интегрируются с другими программными компонентами.

История жизни Python

Python был изобретен примерно в 1990 году Гвидо ван Россумом во время его работы в CWI[1] в Амстердаме. Рептилии здесь не в счет: язык назван в честь комедийного сериала Би-Би-Си «Monty Python's Flying Circus», любителем которого является Гвидо (смотрите следующую глупую врезку). Гвидо занимался распределенной операционной системой Amoeba и языком ABC. В действительности первоначальной мотивацией для Python было создание развитого языка сценариев для системы Amoeba.

Но конструкция Python оказалась настолько общей, что его можно было применять в разнообразных сферах. Сейчас он используется сотнями тысяч специалистов по всему свету во все новых и новых областях. Сегодня компании используют Python в коммерческих продуктах для таких задач, как тестирование микросхем и плат, разработка GUI, поиск в Сети, анимация в фильмах, пользовательская настройка библиотек клас-

Что в имени тебе моем?

Python получил свое название благодаря британскому комедийному телесериалу 1970-х «Monty Python's Flying Circus». Фольклор Python утверждает, что Гвидо ван Россум, создатель Python, смотрел повторные показы этого сериала примерно в то же время, когда ему понадобилось дать имя новому языку, который он разрабатывал. И, как говорят в шоу-бизнесе, «случившееся навсегда вошло в историю».

Такая наследственность служит причиной частого появления в примерах и тексте ссылок на комедийную игру. Например, имя «Spam» особо относится к пользователям Python, а конфронтации иногда называют «Испанской инквизицией». Как правило, когда пользователь Python начинает произносить фразы, не имеющие отношения к реальности, они оказываются заимствованными из сериала или фильмов «Monty Python». Некоторые из этих фраз могут встретиться даже в данной книге. Необязательно, конечно, бежать и брать в прокате «The Meaning of Life» или «The Holy Grail», чтобы делать что-то полезное на Python, но и хуже от этого не будет.

Хотя имя «Python» быстро прижилось, оно послужило причиной некоторых интересных побочных эффектов. Например, когда в 1994 году возникла телеконференция по Python *comp.lang.python*, первые несколько недель она была почти полностью оккупирована людьми, желавшими обсуждать темы, касающиеся телевизионной постановки. Позднее в специальном приложении к журналу *Linux Journal* красовались фотографии Гвидо, облаченного в обязательную «красивую красную форму».

Иногда в конференции Python все еще появляются случайные письма от поклонников сериала. Например, в одном письме невинно предлагалось обмениваться сценариями Monty Python с другими поклонниками. Если бы автор понимал назначение этого форума, то хотя бы указал, выполняются его сценарии под DOS или Unix.

[1] Национальном научно-исследовательском институте математики и информатики Нидерландов. – *Примеч. пер.*

сов C++ и многих других.[1] На самом деле, поскольку Python в полной мере является языком общего назначения, область его применения ограничивается только областью применения компьютеров в целом.

Со времени своего появления в свободном доступе в 1991 году Python продолжает привлекать верных сторонников и в 1994 году породил в Интернет специальную телеконференцию, *comp.lang.python*. А когда в 1995 году писалось первое издание этой книги, дебютировала домашняя страничка Python в WWW на *http://www.python.org*. Она остается официальным местом для всего, связанного с Python.

Для руководства развитием Python со временем образовались организации, нацеленные на поддержку разработчиков Python. В их число входят Python Software Activity (PSA), содействующая конференциям и веб-сайтам по Python, и Python Consortium, образованный организациями, заинтересованными в содействии развитию Python. Когда я пишу эти строки, будущее PSA неясно, но в первые годы она способствовала поддержке Python.

Сегодня Гвидо и несколько других главных разработчиков Python работают в компании под названием Digital Creations и занимаются разработками на Python. Digital Creations, располагающаяся в Вирджинии, является родиной основанного на Python инструментального набора создания веб-приложений Zope (смотрите *http://www.zope.org*). Однако владеет и управляет языком Python независимая организация, и он остается общественно управляемой, подлинной системой open source.

Есть и другие компании, также использующие Python в своих проектах. Например, ActiveState и PythonWare разрабатывают инструменты Python, O'Reilly (издатель данной книги) и компания под названием Foretech организуют ежегодные конференции по Python, а O'Reilly управляет дополнительным веб-сайтом Python (посмотрите Python DevCenter в сети O'Reilly Network на *http://www.oreillynet.com/python*). O'Reilly Python Conference проводится в составе ежегодной конференции по программному обеспечению с открытым кодом (Open Source Software Convention). Хотя в мире профессиональных организаций и компаний изменения происходят чаще, чем в публикуемых книгах, мало сомнений, что язык Python продолжит удовлетворять потребности сообщества его пользователей.

Обязательный список характеристик

Одним из способов описания языка является перечисление его характеристик. Конечно, он станет более осмысленным, когда вы посмотрите на Python в работе; сейчас же мне приходится говорить абстрактно. И то, что делает Python самим собой, это взаимодействие данных характеристик между собой. Но, рассмотрев некоторые атрибуты Python, будет легче дать ему определение. В табл. 1.1 перечислены некоторые наиболее упоминаемые причины привлекательности Python.

Таблица 1.1. Характеристики языка Python

Характеристики	Преимущества
Отсутствие этапов компиляции или компоновки	Короткий цикл разработки
Отсутствие объявлений типов	Более простые, короткие и гибкие программы

[1] Примеры других компаний, использующих Python, есть в предисловии. Более полный список коммерческих приложений можно также найти на *http://www.python.org*.

Таблица 1.1 (продолжение)

Характеристики	Преимущества
Автоматическое управление памятью	Благодаря уборке мусора не нужен код, ведущий учет памяти
Типы данных и операции высокого уровня	Быстрая разработка с помощью встроенных типов объектов
Объектно-ориентированное программирование	Повторное использование кода, интеграция с C++, Java и COM
Встраивание и расширение в C	Оптимизация, подгонка, системный «клей»
Классы, модули, исключительные ситуации	Поддержка модульного программирования
Простой понятный синтаксис и архитектура	Легкость чтения, сопровождения и изучения
Динамическая загрузка модулей C	Упрощенность расширений, меньшие двоичные файлы
Динамическая перезагрузка модулей Python	Можно модифицировать программы, не останавливая их
Универсальная «первоклассная» (first-class) модель объектов	Меньше ограничений и особых правил
Построение программ на этапе исполнения	Кодирование конечным пользователем при возникновении непредвиденных потребностей
Интерактивная динамическая природа	Наращиваемые разработка и тестирование
Доступ к данным интерпретатора	Метапрограммирование, интроспективные объекты
Широкая переносимость интерпретатора	Программирование для разных платформ без отдельных переносов
Компиляция в переносимый байт-код	Скорость выполнения, защита исходного кода
Стандартная переносимая структура GUI	Сценарии Tkinter выполняются на X, Windows и Mac
Поддержка стандартных протоколов Интернета	Легкость доступа к e-mail, FTP, HTTP, CGI и т. д.
Стандартные переносимые системные вызовы	Системные сценарии, безразличные к платформе
Встроенные библиотеки и библиотеки сторонних разработчиков	Большая коллекция готовых программных компонентов
Подлинная открытость исходного кода	Можно бесплатно встраивать и распространять

По правде говоря, в действительности Python является конгломератом функций, заимствованных из других языков. В нем есть элементы, взятые из C, C++, Modula-3, ABC, Icon и других языков. Например, модули пришли в Python из Modula, а опера-

ция среза (slicing) из Icon (по крайней мере, насколько можно вспомнить). А предшествующий опыт Гвидо послужил тому, что Python заимствовал многие идеи ABC, но в него были добавлены и собственные практические возможности, например поддержка расширений на C.

Где хорош Python?

Поскольку области применения Python очень разнообразны, почти невозможно дать надежный ответ на этот вопрос. В общих чертах, любое приложение, которое может выиграть от использования языка, оптимизированного для скорости разработки, может послужить хорошей областью применения Python. С учетом все более сжатых сроков разработки программного обеспечения, это очень широкая категория.

Более точный ответ сформулировать нелегко. Некоторые, например, используют Python как язык встраиваемых расширений, в то время как другие применяют его исключительно в качестве самостоятельного инструмента программирования. В какой-то степени вся данная книга должна ответить на этот самый вопрос: в ней исследуются некоторые наиболее часто выполняемые Python роли. Но сейчас приведем сводку некоторых наиболее распространенных на сегодняшний день способов применения Python:

Системные утилиты

Переносимые инструменты командной строки, тестирование систем.

Сценарии для Интернета

Веб-сайты CGI, апплеты Java, XML, ASP, инструменты электронной почты.

Графические интерфейсы пользователя

С такими API, как Tk, MFC, Gnome, KDE.

Интеграция компонентов

Клиенты библиотек C/C++, настройка продуктов под пользователя.

Доступ к базам данных

Постоянное хранение объектов, интерфейсы баз данных SQL.

Программирование распределенных приложений

С такими API клиент/сервер, как CORBA, COM.

Быстрое создание прототипов и разработка

Одноразовые или поставляемые прототипы.

Языковые модули

Использование Python вместо специализированных синтаксических анализаторов.

И другие

Обработка изображений, численное программирование, ИИ и т. д.

С другой стороны, Python вообще не привязан к какой-либо конкретной области приложений. Например, поддержка интеграции в Python делает его полезным почти в любой системе, которая может выиграть от программируемого оконечного интерфейса. Абстрактно говоря, Python предоставляет сервисы, соединяющие области приложений. Python – это:

- Динамический язык программирования для ситуаций, в которых этап компиляции/сборки невозможен (настройка на месте) или неудобен (создание прототипа, быстрая разработка, системные утилиты)

«Автобусы признаны опасными»

Упоминавшаяся выше организация PSA первоначально была образована как ответ на когда-то давно возникшее в телеконференции Python обсуждение полусерьезного вопроса: «Что будет, если Гвидо попадет под автобус?»

В настоящее время Гвидо ван Россум по-прежнему является верховным арбитром для предложений о внесении в Python изменений, но масса пользователей Python помогает поддерживать язык, работать над расширениями, исправлять ошибки и т. д. Фактически разработка Python сейчас является совершенно открытым процессом: изучать новейшие файлы с исходным кодом или посылать патчи может каждый посетитель веб-сайта (детали смотрите на *http://www.python.org*).

Являясь пакетом с открытым исходным кодом, Python действительно разрабатывается очень большим составом программистов, которые согласованно работают, находясь в разных частях света. С учетом популярности Python нападение со стороны автобуса уже не кажется таким опасным, как раньше, но Гвидо, возможно, считает иначе.

- Мощный, но простой язык, предназначенный для ускоренной разработки и ситуаций, в которых сложность более крупных языков может стать обременительной (прототипирование, кодирование конечным пользователем)
- Обобщенный языковый инструмент для таких ситуаций, в которых иначе потребовалось бы изобретать и реализовывать очередной «язычок» (программируемые системные интерфейсы, инструменты конфигурирования)

При наличии таких общих свойств Python может применяться в любой нужной области путем расширения его библиотеками, относящимися к области применения, встраивания в приложение или самостоятельного использования. Например, роль Python как языка для системных средств обусловлена как его встроенными интерфейсами к сервисам операционной системы, так и самим языком. В действительности, поскольку Python создавался с учетом возможностей интеграции, он естественным образом породил растущую библиотеку расширений и инструментальных средств, доступных разработчикам на Python в виде готовых компонентов. В табл. 1.2 перечислены лишь некоторые из них, о большинстве подробнее рассказывается в данной книге или на веб-сайте Python.

Таблица 1.2. Некоторые популярные инструментальные средства и расширения Python

Область применения	Расширения
Системное программирование	Сокеты, потоки, сигналы, конвейеры, вызовы RPC, интерфейс к POSIX
Графические интерфейсы пользователя	Tk, PMW, MFC, X11, wxPython, KDE, Gnome
Интерфейсы баз данных	Oracle, Sybase, PostGres, mSQL, постоянное хранение, dbm
Инструменты Microsoft Windows	MFC, COM, ActiveX, ASP, ODBC, .NET
Инструменты Интернета	JPython, средства CGI, анализаторы HTML/XML, средства e-mail, Zope

Таблица 1.2 (продолжение)

Область применения	Расширения
Распределенные объекты	DCOM, CORBA, ILU, Fnorb
Другие популярные инструменты	SWIG, PIL, регулярные выражения, NumPy, криптография

Для чего Python не годится?

Опять-таки, если говорить честно, некоторые задачи находятся за пределами возможностей Python. Как и все динамические языки, Python (в настоящей его реализации) не настолько быстр и эффективен, как статические компилируемые языки типа C. Во многих областях эта разница не имеет значения: в программах, занятых в основном взаимодействием с пользователем или передачей данных по сетям, производительности Python обычно более чем достаточно для удовлетворения потребностей всего приложения. Но в некоторых областях производительность сохраняет первостепенное значение.

Поскольку в настоящее время Python является интерпретируемым языком,[1] одного только Python обычно оказывается недостаточно для создания компонентов, критическим свойством которых является быстродействие. Операции, интенсивно выполняющие вычисления, можно реализовывать как компилированные *расширения* (*extensions*) Python и кодировать на языке низкого уровня типа C. Python нельзя использовать в качестве единственного языка, на котором реализуются такие компоненты, но он хорошо работает в качестве клиентского интерфейса сценариев для них.

Например, поддержка численного программирования и обработки изображений введена в Python путем объединения оптимизированных расширений с интерфейсом языка Python. В такой системе после того, как разработаны оптимизированные расширения, программирование происходит в основном на более высоком уровне сценариев Python. В итоге получается инструмент численного программирования, который одновременно эффективен и прост в использовании.

Кроме того, в таких областях применения Python по-прежнему можно использовать для создания прототипов. Система может быть сначала реализована на Python, а затем перед поставкой полностью или частично переведена на такой язык, как C. У C и Python есть свои сильные стороны и задачи. Гибридный подход, при котором C используется для модулей с интенсивными вычислениями, а Python для создания прототипов и клиентских интерфейсов, может усилить выгоды от использования обоих языков.

В некотором смысле, Python находит компромисс между эффективностью и гибкостью тем, что вообще не ищет его. Есть язык, оптимизированный для простоты ис-

[1] Python «интерпретируется» таким же образом, как Java: исходный код Python автоматически компилируется (транслируется) в промежуточный формат, называемый «байт-код», который выполняется затем виртуальной машиной Python (то есть системой исполнения Python). Благодаря этому сценарии Python лучше переносятся и выполняются быстрее, чем чистый интерпретатор, который выполняет исходный или древовидный код. Но в результате Python также становится более медленным, чем подлинные компиляторы, транслирующие исходный код в двоичный машинный код для конкретного ЦП. Имейте, однако, в виду, что эти детали касаются стандартной реализации Python; JPython (или. «Jython») компилирует сценарии Python в байт-код Java, а новая реализация для C#/.NET компилирует сценарии Python в двоичные *.exe*-файлы. Оптимизирующий компилятор Python может разрушить все высказанные в данной главе опасения относительно производительности (будем на это надеяться).

пользования, и есть средства интеграции его с другими языками. Сочетая компоненты, написанные на Python и компилируемых языках типа C и C++, разработчики могут выбирать нужную пропорцию простоты использования и эффективности для каждого конкретного случая. Хотя навряд ли Python когда-либо станет таким же быстрым, как C, в большинстве сегодняшних программных проектов скорость разработки, обеспечиваемая Python, не менее важна, чем скорость выполнения, обеспечиваемая C.

О честности в рекламе

В конце книги мы вернемся к некоторым важнейшим идеям, представленным в этой главе, после того как изучим Python в действии. Однако я хочу заранее подчеркнуть, что занимаюсь информатикой, а не маркетингом. Я намерен соблюдать в данной книге предельную откровенность как в отношении достоинств Python, так и его недостатков. Несмотря на то что Python является одним из самых простых в использовании языков программирования из всех когда-либо созданных, некоторые ловушки действительно существуют, и о них будет говориться в книге.

Сразу и начнем. Самая большая ловушка, о которой, возможно, следует знать, такова: *с помощью Python чрезвычайно легко быстро создать плохую конструкцию*. Это действительно проблема. Поскольку писать программы на Python так просто и быстро в сравнении с обычными языками, очень легко увлечься актом программирования, уделяя недостаточно внимания самой задаче, которую на самом деле нужно решить.

В действительности Python может оказаться явно соблазнительным – настолько, что потребуется сознательно противостоять искушению быстро набросать на Python программу, которая работоспособна и, как можно утверждать, «крута», но с ней вы окажетесь столь же далеко от сопровождаемой реализации первоначального замысла, как вы были в начале работы. Естественные задержки, присущие разработке на компилируемом языке, – исправление ошибок, выявленных компилятором, компоновка библиотек и тому подобное, – отсутствуют при работе с Python и не помогут вам притормозить и подумать.

Не всегда все так плохо. В большинстве случаев первые наброски на скорую руку послужат переходу к лучшим решениям, которые вы впоследствии сохраните. Но имейте в виду: даже такой язык быстрой разработки приложений, как Python, *не может заменить необходимость думать*. Прежде чем начать вводить код, лучше сесть и подумать. По крайней мере, на сегодняшний день ни одному языку программирования не удалось сделать ненужным интеллект.

I

Системные интерфейсы

В этой первой технической части книги представлены инструменты Python для системного программирования – интерфейсы к сервисам базовой операционной системы, а также к контексту выполнения программы. Часть I состоит из следующих глав:

- Глава 2 «Системные инструменты». Эта глава всесторонне рассматривает часто используемые инструменты системных интерфейсов и показывает, как работать с потоками, файлами, каталогами, аргументами командной строки, переменными оболочки и многим другим. Глава ведет изложение неторопливо и частично предназначена служить справочником.
- Глава 3 «Системные средства параллельного выполнения». Служит введением в поддержку библиотекой Python параллельного выполнения программ. Здесь вы найдете описание потоков выполнения (нитей), ветвления процессов, каналов, сигналов и тому подобного.
- Глава 4 «Более крупные системные примеры, часть 1» и глава 5 «Более крупные системные примеры, часть 2». В этих главах собраны типичные примеры системного программирования, основанные на материале первых двух глав. Среди прочего, представленные здесь сценарии Python демонстрируют, как разрезать и соединять файлы, сравнивать и копировать каталоги, генерировать веб-страницы по шаблонам и запускать программы и веб-броузеры переносимым образом. Вторая из глав сосредоточена на более сложных примерах файлов и каталогов; в первой представлены разнообразные конкретные примеры использования системных инструментов.

Хотя в этой части книги особое значение придается задачам системного программирования, показанные в ней средства являются универсальными и часто используются в последующих главах.

2

Системные инструменты

«os.path – дорога к Знанию»

В этой главе начинается рассмотрение способов применения Python для решения практических задач программирования. Здесь и далее мы изучим, как использовать Python для написания системных инструментов, графических интерфейсов пользователя, приложений баз данных, сценариев Интернета, веб-сайтов и многого другого. Попутно будут также изучаться в действии важнейшие концепции программирования на Python: повторное использование кода, сопровождаемость, объектно-ориентированное программирование и т. д.

В первой части этой книги мы начинаем путешествие по программированию на Python с исследования *области системных приложений* – сценариев, работающих с файлами, программами и окружением программ в целом. Хотя примеры из этой области сконцентрированы на определенном роде задач, используемая в них технология окажется также полезной и в последующих частях книги. Иными словами, следует пуститься в путь с этого места, если только вы уже не являетесь знатоком системного программирования на Python.

Зачем здесь нужен Python?

Системные интерфейсы Python охватывают области приложений, но в последующих четырех главах большинство примеров будет относиться к категории *системных инструментов* – программ, иногда называемых утилитами командной строки, сценариями оболочки или некоторыми сочетаниями этих слов. Независимо от названия, вы, вероятно, уже знакомы со сценариями этого типа: они выполняют такие задачи, как обработка файлов в каталоге, запуск тестовых сценариев и тому подобное. Исторически такие программы писались на непереносимых и синтаксически неясных языках оболочек, таких как командные файлы DOS, csh и awk.

Однако даже в этой относительно простой области ярко проявляются лучшие свойства Python. Например, простота использования Python и обширные встроенные библиотеки упрощают (и даже делают приятным) использование развитых системных инструментов, таких как потоки, сигналы, ветвление процессов, сокеты и аналогичные; доступ к этим средствам значительно труднее осуществить в неясном синтаксисе языков оболочек и долгих циклах разработки компилируемых языков. Поддержка в Python таких идей, как ясность кода и объектно-ориентированное программирование, способствует написанию таких инструментов оболочки, которые можно читать, сопровождать и многократно использовать. При использовании Python нет необходимости начинать с нуля каждый новый сценарий.

Более того, мы обнаружим, что в Python не только есть все интерфейсы, необходимые для написания системных инструментов, но он поддерживает *переносимость* сценариев. При использовании стандартной библиотеки Python большинство написанных на Python системных сценариев становится автоматически переносимым на все основные платформы. Сценарий для обработки каталогов, написанный, к примеру, под Windows, обычно может выполняться и под Linux без всякой правки кода: нужно просто скопировать исходный код. При разумном использовании Python может стать единственным средством, которое необходимо знать для создания системных сценариев.

«Батарейки – в комплекте»

В данной главе и следующих за ней речь идет одновременно о языке Python и его стандартной библиотеке. Хотя Python и представляет собой легкий язык написания сценариев, в реальных разработках на Python используется обширная библиотека средств программирования (по последним подсчетам около 200 модулей), поставляемых вместе с пакетом Python. В действительности стандартные библиотеки настолько мощны, что нередко можно слышать в отношении Python фразу «batteries included» (батарейки – в комплекте), обычно приписываемую Фрэнку Стаяно (Frank Stajano) и означающую, что все необходимое для реальной повседневной деятельности уже готово и может быть импортировано.

Как будет показано, стандартные библиотеки составляют существенную часть задачи программирования на Python. Овладев базовым языком, вы заметите, что в основном заняты применением встроенных функций и модулей, поставляемых вместе с системой. Но самое интересное происходит в библиотеках. На практике интереснее всего программы становятся тогда, когда в них начинают использоваться службы, находящиеся вне интерпретатора языка: сети, файлы, GUI, базы данных и т. д. Все это поддерживается стандартной библиотекой Python – собранием готовых модулей, написанных на Python и C, – которая устанавливается вместе с интерпретатором Python.

Помимо стандартной библиотеки Python существуют дополнительные пакеты для Python производства сторонних разработчиков, которые могут быть получены и установлены отдельно. Когда писалась эта книга, большинство таких расширений сторонних разработчиков можно было найти путем поиска и ссылок на *http://www.python.org* и сайтах Python «Starship» и «Vaults of Parnassus» (на которые можно также попасть по ссылкам с *http://www.python.org*). Если нужно сделать с помощью Python что-либо особенное, есть большая вероятность найти бесплатный модуль с открытым исходным кодом, который поможет решить задачу. Большинство инструментов, используемых в этой книге, входит в стандартный дистрибутив Python, а то, что должно устанавливаться отдельно, будет отмечено особо.

Обзор системных сценариев

В следующих двух разделах будет сделан беглый обзор модулей `sys` и `os`, а затем изложение перейдет к важным понятиям системного программирования. Я не намерен демонстрировать каждый элемент в каждом встроенном модуле, а прежде всего хочу показать, как самостоятельно получить дополнительные сведения. Такая задача служит и официальным извинением тому, что представляются немногие базовые идеи системных концепций, попутно мы напишем код первого сценария для форматирования документации.

Системные модули Python

Большинство интерфейсов Python системного уровня находится лишь в двух модулях: sys и os. Это некоторое упрощение: к данной области относятся и другие стандартные модули (например, glob, socket, thread, time, fcntl). Системными интерфейсами являются, в действительности, и некоторые встроенные функции (например, open). Но sys и os вместе образуют ядро арсенала системных инструментов Python.

По крайней мере в принципе, sys экспортирует компоненты, относящиеся к самому *интерпретатору* Python (например, путь поиска модулей), а os содержит переменные и функции, соответствующие операционной системе, в которой выполняется Python. На практике это различие может быть не столь отчетливым (например, стандартные потоки ввода и вывода находятся в sys, но можно утверждать, что они связаны с парадигмами операционной системы). Могу вас обрадовать: инструменты, находящиеся в этих модулях, будут использоваться так часто, что их местонахождение прочно отпечатается в вашей памяти.[1]

Модуль os пытается также предоставить *переносимый* интерфейс программирования для используемой операционной системы: его функции могут быть по-разному реализованы для различных платформ, но в сценариях Python они выглядят одинаково. Кроме того, модуль os экспортирует вложенный подмодуль os.path, предоставляющий переносимый интерфейс к средствам обработки файлов и каталогов.

Источники документации по модулям

Как можно было заключить из предыдущих параграфов, чтобы научиться писать системные сценарии на Python, нужно в основном изучить системные модули Python. К счастью, существует множество источников информации, облегчающих эту задачу, – от атрибутов модуля до печатных справочников и книг.

Например, если нужно узнать, какие элементы экспортирует модуль, можно прочесть соответствующий раздел руководства по библиотеке, исследовать его исходный код (все-таки Python является программным обеспечением с открытым кодом) или получить список атрибутов и строку документации в интерактивном режиме. Сделаем импорт sys и посмотрим, что в нем находится:

```
C:\...\PP2E\System> python
>>> import sys
>>> dir(sys)
['__doc__', '__name__', '__stderr__', '__stdin__', '__stdout__', 'argv',
'builtin_module_names', 'copyright', 'dllhandle', 'exc_info', 'exc_type',
'exec_prefix', 'executable', 'exit', 'getrefcount', 'hexversion', 'maxint',
'modules', 'path', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval',
'setprofile', 'settrace', 'stderr', 'stdin', 'stdout', 'version', 'winver']
```

Функция dir просто возвращает список строк с именами всех атрибутов для любого объекта, имеющего атрибуты; это удобный способ вспомнить содержимое модуля, находясь в интерактивной подсказке. Видно, например, что существует нечто с именем sys.version, поскольку имя version возвращено в результатах dir. Если этого недостаточно, всегда можно обратиться к строке __doc__ встроенного модуля:

---

[1] Они могут проникать и в ваше подсознание. Новички Python иногда появляются на форумах с выражениями радости по поводу того, что впервые «видели сны на Python». Оставляя в стороне фрейдистские интерпретации, не так плохо, если сны снятся на Python, ведь есть много значительно более плохих языков.

```
>>> sys.__doc__
...
...много текста удалено отсюда...
...
count for an object (plus one :-)\012setcheckinterval() -- control how often
the interpreter checks for events\012setprofile() -- set the global profiling
function\012settrace() -- set the global debug tracing function\012"
```

Постраничный вывод строк документации

Встроенный атрибут `__doc__` обычно содержит строку документации, которая может выглядеть несколько странно при выводе: это одна длинная строка с символами перевода строки, выводящимися как `\012`, а не красивый список строк. Чтобы отформатировать эти строки для более человеческого изображения, я обычно использую вспомогательный сценарий типа представленного в примере 2.1.

Пример 2.1. PP2E\System\more.py

```
##############################################################################
# расщепить и интерактивно выводить постранично строку или текстовый файл;
##############################################################################
import string
def more(text, numlines=15):
    lines = string.split(text, '\n')
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print line
        if lines and raw_input('More?') not in ['y', 'Y']: break
if __name__ == '__main__':
    import sys                          # если выполняется, а не импортируется
    more(open(sys.argv[1]).read(), 10)  # содержимое страницы файла в командной строке
```

Главной в этом файле является функция `more`, и если вы хоть что-то знаете о Python, она должна быть достаточно понятна: просто строка расщепляется в ней по символам конца строки, а затем делается срез и выводится сразу несколько строк (по умолчанию 15), чтобы на экране не было прокрутки. Выражение среза `lines[:15]` дает первые 15 элементов списка, а `lines[15:]` дает остальные; чтобы показывать каждый раз иное число строк, передайте его в аргументе `numlines` (например, в последней строке примера 2.1 в аргумент `numlines` функции `more` передается 10).

Вызов встроенной функции `string.split`, используемый в этом сценарии, возвращает список подстрок (например, `["line", "line",...]`). Как будет показано далее в этой главе, символом конца строки в сценарии Python всегда является `\n` (что в восьмеричном виде представляется как `\012`) вне зависимости от платформы. (Если вы еще не знаете, почему это имеет значение: символы DOS `\r` отбрасываются при чтении.)

Знакомство с модулем string

Это простая программа Python, но в ней представлены три важные темы, заслуживающие краткого упоминания: она использует модуль `string`, осуществляет чтение из файла, и сделана так, что может быть импортирована или выполнена. Модуль Python `string` как таковой не является системным инструментом, но действует в большинстве программ Python. В действительности он будет постоянно встречаться на протяжении этой и последующих глав, поэтому дадим краткий обзор наиболее полезных экспортируемых им элементов. В модуле `string` есть вызовы для поиска и замены:

```
>>> import string
>>> string.find('xxxSPAMxxx', 'SPAM')           # возвращает первое смещение
3
>>> string.replace('xxaaxxaa', 'aa', 'SPAM')    # глобальная замена
'xxSPAMxxSPAM'

>>> string.strip('\t  Ni\n')                    # удаление пробельных символов
'Ni'
```

Вызов `string.find` возвращает смещение первого вхождения подстроки, а `string.replace` осуществляет глобальный поиск и замену. В этом модуле подстроки являются просто строками; в главе 18 «Текст и язык» будут представлены модули, которые позволяют использовать при поиске и замене *шаблоны* регулярных выражений. В модуле `string` есть также константы и функции, полезные, например, для преобразования регистра:

```
>>> string.lowercase                            # константы регистра, конвертеры
'abcdefghijklmnopqrstuvwxyz'

>>> string.lower('SHRUBBERRY')
'shrubberry'
```

Существуют также средства, позволяющие расщеплять строки по разделителю, который является подстрокой, и соединять их вместе, вставляя между ними подстроку. Эти средства будут изучены далее в этой книге, но в качестве знакомства покажем, как они работают:

```
>>> string.split('aaa+bbb+ccc', '+')           # расщепить в список подстрок
['aaa', 'bbb', 'ccc']
>>> string.split('a b\nc\nd')                   # разделитель по умолчанию: пробельный символ
['a', 'b', 'c', 'd']

>>> string.join(['aaa', 'bbb', 'ccc'], 'NI')    # объединить подстроки из списка
'aaaNIbbbNIccc'
>>> string.join(['A', 'dead', 'parrot'])        # разделитель по умолчанию: пробел
'A dead parrot'
```

Эти вызовы оказываются удивительно мощными. Например, строку колонок данных, разделенных символами табуляции, можно разобрать по колонкам за один вызов `split`; сценарий *more.py* использует его, чтобы разбить строку в список строк. В действительности можно эмулировать вызов `string.replace` с помощью комбинации split/join:

```
>>> string.join(string.split('xxaaxxaa', 'aa'), 'SPAM')   # замена сложным путем
'xxSPAMxxSPAM'
```

Запомните на будущее, что Python не преобразует автоматически строки в числа и наоборот; если нужно использовать те или другие, необходимо указать это с помощью ручных преобразований:

```
>>> string.atoi("42"), int("42"), eval("42")     # преобразование строки в int
(42, 42, 42)

>>> str(42), '42', ("%d" % 42)                   # преобразование int в строки
('42', '42', '42')

>>> "42" + str(1), int("42") + 1                 # конкатенация, сложение
('421', 43)
```

В последней приведенной команде первое выражение вызывает конкатенацию строк (так как оба операнда являются строками), а второе вызывает сложение целых чисел

(поскольку оба объекта являются числами). Python не делает предположений, что вы имели в виду то или иное, и не выполняет автоматического преобразования: на практике Python старается по возможности избегать чудес. Более подробно о средствах для работы со строками будет рассказано далее в этой книге (в действительности им посвящена целая глава в части IV «Разные темы»), но обязательно поищите также в руководстве по библиотеке дополнительные инструменты модуля `string`.

Начиная с Python 1.6 у *объектов* `string` появились *методы*, соответствующие функциям *модуля* `string`. Например, если есть объект `string` с именем `X`, то `X.split()` теперь выполняет то же, что `string.split(X)`. Для примера 2.1 это означает эквивалентность следующих строк:

```
lines = string.split(text, '\n')
lines = text.split('\n')
```

но для последнего формата не требуется конструкции import. Модуль `string` в обозримом будущем никуда не исчезнет, но методы `string` будут, вероятно, новым увлечением в области обработки текста на Python.

Основы операций с файлами

Сценарий *more.py* также может открывать внешний файл, имя которого задается в командной строке, с помощью встроенной функции `open` и целиком считывает его в память с помощью метода `read` объекта файла. Поскольку объекты файлов, возвращаемые `open`, являются составной частью самого базового языка Python, предполагается, что у читателя есть хотя бы беглое знакомство с ними. Но в случае, если эта глава попалась вам в самом начале изучения Python, укажем, что вызовы:

```
open('file').read()          # прочесть весь файл в строку
open('file').read(N)         # прочесть очередные N байтов в строку
open('file').readlines()     # прочесть весь файл в список строк
open('file').readline()      # прочесть очередную строку до '\n'
```

загружают в строку содержимое файла, загружают в строку набор байтов фиксированной длины, загружают содержимое файла в список строк и загружают в строку очередную строку файла, соответственно. Как мы скоро увидим, эти вызовы можно также применять в Python к командам оболочки. У объектов файлов есть также методы `write`, которые посылают строки в соответствующий файл. Более глубоко темы, связанные с файлами, раскрываются далее в этой главе.

Два способа использования программ

Последние несколько строк в *more.py* знакомят также с одним из первых важных понятий в программировании инструментов оболочки. Они настраивают файл так, чтобы его можно было использовать двумя способами: как *сценарий* или как *библиотеку*. В каждом модуле Python есть встроенная переменная `__name__`, которую Python устанавливает равной `__main__`, только если файл выполняется как программа, а не импортируется в качестве библиотеки. Из-за этого функция `more` в этом файле автоматически выполняется в последней строке файла, когда сценарий выполняется в качестве основной программы, а не импортируется в какое-либо другое место. С помощью этого простого приема реализуется способ написания многократно используемого кода сценария: благодаря кодированию логики программы в виде *функции*, а не

в виде кода верхнего уровня, ее можно импортировать и повторно использовать в других сценариях.

В результате появляется возможность выполнять *more.py* самостоятельно или импортировать его и вызывать функцию more из любого другого места. При запуске файла как программы верхнего уровня в командной строке указывается имя файла, который нужно прочесть и выводить постранично: ниже в этой главе будет полностью описано, как слова, вводимые в команде для запуска программы, появляются в Python во встроенном списке sys.argv. Вот, например, файл сценария, который постранично выводит сам себя (обязательно вводите эту командную строку из каталога *PP2E\System*, иначе входной файл не будет найден; причина этого будет пояснена позднее):

```
C:\...\PP2E\System>python more.py more.py
################################################################################
# расщепить и интерактивно выводить постранично строку или текстовый файл;
################################################################################

import string

def more(text, numlines=15):
    lines = string.split(text, '\n')
    while lines:
        chunk = lines[:numlines]
More?y
        lines = lines[numlines:]
        for line in chunk: print line
        if lines and raw_input('More?') not in ['y', 'Y']: break

if ____name____ == '__main__':
    import sys                          # если выполняется, а не импортируется
    more(open(sys.argv[1]).read(), 10)  # содержимое страницы файла в командной строке
```

Когда файл *more.py* импортируется, в его функцию more неявно передается строка, и как раз такая утилита нужна нам для текста документации. Запуск этой утилиты со строкой документации модуля sys дает нам немножко больше информации о том, какие возможности даются сценариям, в виде, пригодном для чтения:

```
>>> from more import more
>>> more(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
exitfunc -- you may set this to a function to be called when Python exits
stdin -- standard input file object; used by raw_input() and input()
stdout -- standard output file object; used by the print statement
stderr -- standard error object; used for error messages
  By assigning another file object (or an object that behaves like a file)
  to one of these, it is possible to redirect all of the interpreter's I/O.
More?
```

Нажатие «y» (и клавиши Enter) заставляет функцию отобразить несколько следующих строк документации и снова вывести приглашение, если список строк еще не закончился. Попробуйте сделать это у себя на машине, и вы увидите, как выглядит оставшаяся часть строки документации.

Руководства по библиотекам Python

Если полученных подробностей все еще недостаточно, то следующим шагом будет чтение в руководстве по библиотекам Python раздела, посвященного `sys`, с целью получения полной информации. Все стандартные руководства Python поставляются в виде страниц HTML, чтобы их можно было прочесть в любом броузере, имеющемся на компьютере. Они есть на CD, прилагаемом к данной книге, и устанавливаются в Windows вместе с Python, однако вот несколько простых указаний:

- В Windows щелкните по кнопке Start, выберите Programs, выделите пункт Python и выберите пункт Python Manuals. Руководства чудесным образом появятся на вашем экране в окне броузера, например, Internet Explorer.
- В Linux можно щелкнуть по элементам для руководств в менеджере файлов или запустить броузер из командной строки и переместиться в то место, где на вашей машине находятся файлы HTML руководства.
- Если у вас на машине руководств не обнаружилось, всегда можно прочесть их в Интернете. Перейдите на веб-сайт Python *http://www.python.org* и найдите ссылки, ведущие к документации.

В любом случае выберите руководство «Library», если вас интересуют такие вещи, как `sys`. Стандартное руководство Python содержит также краткий учебник, справочник по языку, справочники по расширениям и многое другое.

Коммерческие справочники

Рискуя заслужить упрек за рекламу в книге, я должен упомянуть, что можно приобрести комплект руководств по Python, отпечатанный и переплетенный; подробности и ссылки можно найти на информационной странице по изданиям на *http://www.python.org*. На сегодняшний день есть также коммерческие печатные справочники по Python, в том числе «Python Essential Reference» (New Riders Publishing) и «Python Pocket Reference» (O'Reilly). Первый из них полнее и содержит примеры, однако последний удобно использовать, чтобы освежить в памяти, когда вы уже раз-другой изучили библиотеку.[1] Следите также за выходом готовящейся к печати в O'Reilly «Python Standard Library».

Модуль sys

Перейдем к подробностям, касающимся модуля. Как говорилось выше, модули `sys` и `os` образуют ядро набора инструментов Python для системных задач. Сделаем сейчас краткий интерактивный обзор некоторых инструментов, имеющихся в этих двух модулях, прежде чем использовать их в более крупных примерах.

Платформы и версии

Как и в большинстве модулей, в `sys` есть атрибуты, содержащие информацию, и функции, выполняющие действия. Например, в его атрибутах можно найти название опе-

[1] Я написал последний в качестве замены справочному приложению, имевшемуся в первом издании книги; он должен явиться дополнением к тексту, который вы читаете. Однако поскольку его автором являюсь я, то больше ничего здесь не скажу... за исключением того, что вам следует при очередном посещении книжного магазина приобрести по экземпляру для друзей, коллег по работе, старых друзей по колледжу и всех близких и дальних родственников (шучу, конечно).

рационной системы, в которой выполняется код, наибольшее целое число, допустимое на данной машине, и номер версии интерпретатора Python, выполняющего код:

```
C:\...\PP2E\System>python
>>> import sys
>>> sys.platform, sys.maxint, sys.version
('win32', 2147483647, '1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)]')
>>>
>>> if sys.platform[:3] == 'win': print 'hello windows'
...
hello windows
```

Если код должен по-разному выполняться на разных машинах, просто проверьте строку `sys.platform`, как сделано в этом примере: хотя Python по большей части независим от платформы, а непереносимые средства обычно заключаются в проверки `if` типа той, что здесь приведена. Например, далее будет показано, что средства запуска программ и взаимодействия на низком уровне с консолью различаются в зависимости от платформы, поэтому проверьте `sys.platform`, чтобы выбрать правильное средство для той машины, на которой выполняется сценарий.

Путь для поиска модулей

Модуль `sys` еще позволяет проверить пути поиска модулей как в интерактивном режиме, так и из программы Python. `sys.path` дает список строк, представляющих действительный путь поиска в выполняющемся интерпретаторе Python. Когда импортируется модуль, Python просматривает этот путь слева направо, разыскивая файл модуля в каждом каталоге, указанном в списке. Поэтому здесь нужно проверять, что путь поиска действительно задан так, как нужно.[1]

Список `sys.path` просто инициализируется при первом запуске интерпретатора из PYTHONPATH с добавлением системных значений по умолчанию. В действительности обнаружится довольно много каталогов, которые не входят в ваш PYTHONPATH, если посмотреть на `sys.path` в интерактивном режиме: в него входит также указатель на исходный каталог сценария (пустая строка – подробнее я объясню это далее после знакомства с `os.getcwd`) и набор каталогов стандартных библиотек, который может быть различным в каждой установке:

```
>>> sys.path
['', 'C:\\PP2ndEd\\examples', ...плюс еще стандартные пути... ]
```

Как ни удивительно, `sys.path` можно *изменять* программным образом: в сценарии могут выполняться такие действия над списками, как `append`, `del` и другие, с помощью которых путь поиска настраивается на этапе исполнения. Python всегда использует для импорта текущую установку `sys.path` независимо от того, как вы ее изменили:

```
>>> sys.path.append(r'C:\mydir')
>>> sys.path
['', 'C:\\PP2ndEd\\examples', ...другие стандартные пути... , 'C:\\mydir']
```

Такое непосредственное изменение `sys.path` является альтернативой установке переменной оболочки PYTHONPATH, но не очень хорошей: изменения в `sys.path` сохраня-

[1] Может случиться, что Python видит PYTHONPATH иначе, чем вы. Синтаксическая ошибка в файлах конфигурации оболочки может испортить установку PYTHONPATH, даже если она кажется вам нормальной. Например, в Windows при наличии пробелов вокруг = в команде DOS `set` в файле `autoexec.bat` (например, `set NAME = VALUE`) `NAME` в действительности получит в качестве значения пустую строку, а не `VALUE`!

ются лишь до завершения процесса Python, и их нужно повторять заново при каждом новом запуске программы или сеанса Python.

Таблица загруженных модулей

В модуле sys есть также ловушки (hooks) для интерпретатора; sys.modules, например, служит словарем, в котором есть по одной записи name:module для каждого модуля, импортированного в сеанс или программу Python (точнее, в процесс, вызвавший Python):

```
>>> sys.modules
{'os.path': <module 'ntpath' from 'C:\Program Files\Python\Lib\ntpath.pyc'>,...

>>> sys.modules.keys()
['os.path', 'os', 'exceptions', '__main__', 'ntpath', 'strop', 'nt', 'sys',
'__builtin__', 'site', 'signal', 'UserDict', 'string', 'stat']
>>>
>>> sys
<module 'sys' (built-in)>
>>> sys.modules['sys']
<module 'sys' (built-in)>
```

Такую ловушку можно использовать для создания программ, выводящих или иным образом обрабатывающих все модули, загруженные программой (нужно просто пройти по списку ключей sys.modules). sys экспортирует также средства для получения счетчика ссылок на объект, используемого сборщиком мусора Python (getrefcount), проверки того, какие модули встроены в Python (builtin_module_names), и других вещей.

Пути каталогов в Windows

Поскольку *обратный слэш (обратная косая черта)* в строке Python обычно означает начало escape-последовательности, пользователи Windows должны следить за тем, чтобы удваивать обратную косую черту при использовании ее в строках с путями каталогов DOS (например, в "C:\\dir" в действительности \\ является escape-последовательностью, означающей \) или использовать константы необрабатываемых строк (*raw string*), чтобы сохранять обратную косую черту литерально (например, r"C:\dir").

При просмотре путей каталогов под Windows (как sys.path в листинге диалога) Python выводит \\ в смысле одной \. Формально можно обойтись в строке одной \, если за ней следует символ, не воспринимаемый Python как продолжение escape-последовательности, но использовать удвоение и необрабатываемые строки обычно легче, чем запоминать таблицы escape-кодов.

Обратите также внимание на то, что большинство вызовов библиотеки Python в качестве разделителей путей каталогов принимает как прямую (/), так и обратную (\) косую черту независимо от используемой платформы. Это значит, что / обычно действует и в Windows, что способствует созданию сценариев, переносимых на Unix. Описываемые далее в этой главе инструменты из модулей os и os.path также способствуют переносимости путей в сценариях.

Сведения об исключительных ситуациях

Некоторые атрибуты модуля sys позволяют получить все сведения о самой последней исключительной ситуации (exception, в дальнейшем для краткости будем использовать термин «исключение»), обработанной Python. Это удобно, когда требуется обра-

батывать исключения в более общих случаях. Например, функция sys.exc_info возвращает тип, значение и объект обратной трассировки (traceback) самой последней исключительной ситуации:

```
>>> try:
...     raise IndexError
... except:
...     print sys.exc_info()
...
(<class exceptions.IndexError at 7698d0>, <exceptions.IndexError instance at
797140>, <traceback object at 7971a0>)
```

Эту информацию можно использовать для форматирования собственного сообщения об ошибке, выводимого во всплывающем окне GUI или веб-странице HTML (вспомните, что по умолчанию не перехваченные исключения завершают программы с выводом сообщения об ошибке Python). Замечание о переносимости: тип, значение и объект обратной трассировки (traceback) самой последней исключительной ситуации доступны и под другими именами:

```
>>> try:
...     raise TypeError, "Bad Thing"
... except:
...     print sys.exc_type, sys.exc_value
...
exceptions.TypeError Bad Thing
```

Но эти имена представляют одиночную глобальную исключительную ситуацию и не специфичны для конкретного *потока* (о потоках рассказывается в следующей главе). Если вы хотите возбуждать и перехватывать исключения в нескольких потоках, то подробности об исключении, связанном с потоком, предоставляет exc_info().

Другие элементы, экспортируемые модулем sys

Модуль sys экспортирует и другие средства, с которыми мы встретимся в контексте более крупных тем и примеров далее в этой главе и книге. Например:

- *Аргументы командной строки* можно получить в виде списка строк под именем sys.argv
- *Стандартные потоки* доступны в виде stdin, stdout и stderr
- *Завершение программы* можно вызвать с помощью sys.exit

Однако поскольку эти вещи приводят к более крупным темам, о них будет рассказано ниже в отдельных разделах этой и последующих глав.

Модуль os

Как уже говорилось, в os содержатся все обычные вызовы операционной системы, с которыми вы могли ранее встречаться в своих программах на C и сценариях оболочки. Его вызовы имеют дело с каталогами, процессами, переменными оболочки и т. д. Формально этот модуль предоставляет инструментальные средства *POSIX* – переносимого стандарта вызовов операционной системы – вместе с платформо-независимыми средствами работы с каталогами, к которым относится вложенный модуль os.path. Действует os как в значительной мере переносимый интерфейс к системным вызовам компьютера: обычно сценарии, написанные с использованием os и os.path, могут выполняться на любой платформе без внесения изменений.

В действительности, если посмотреть исходный код модуля os, можно обнаружить, что на самом деле он просто импортирует тот специфический для платформы системный модуль, который установлен у вас на машине (например, nt, mac, posix). Взгляните на файл *os.py* в каталоге с исходным кодом библиотеки Python – он просто выполняет команду from* для копирования всех имен из специфического для платформы модуля. Однако благодаря тому, что всегда импортируется os, а не специфические для платформы модули, сценарии становятся лучше устойчивыми к различиям в реализациях для разных платформ.

Большие списки os

Бросим беглый взгляд на основные интерфейсы в os. Если изучать атрибуты этого модуля в интерактивном режиме, получится громадный список имен, который будет различным для разных версий Python, скорее всего будет зависеть от платформы и не будет слишком полезен, если не изучить, что означает каждое имя:

```
>>> import os
>>> dir(os)
['F_OK', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_RDONLY', 'O_RDWR',
'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO',
'P_OVERLAY', 'P_WAIT', 'R_OK', 'UserDict', 'W_OK', 'X_OK', '_Environ',
'__builtins__', '__doc__', '__file__', '__name__', '_execvpe', '_exit',
'_notfound', 'access', 'altsep', 'chdir', 'chmod', 'close', 'curdir',
'defpath', 'dup', 'dup2', 'environ', 'error', 'execl', 'execle', 'execlp',
'execlpe', 'execv', 'execve', 'execvp', 'execvpe', 'fdopen', 'fstat', 'getcwd',
'getpid', 'i', 'linesep', 'listdir', 'lseek', 'lstat', 'makedirs', 'mkdir',
'name', 'open', 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read',
'remove', 'removedirs', 'rename', 'renames', 'rmdir', 'sep', 'spawnv',
'spawnve', 'stat', 'strerror', 'string', 'sys', 'system', 'times', 'umask', 'unlink', 'utime',
'write']
```

Помимо этого, вложенный модуль os.path экспортирует дополнительные инструменты, большинство из которых связано с переносимой обработкой имен файлов и каталогов:

```
>>> dir(os.path)
['__builtins__', '__doc__', '__file__', '__name__', 'abspath', 'basename',
'commonprefix', 'dirname', 'exists', 'expanduser', 'expandvars', 'getatime',
'getmtime', 'getsize', 'isabs', 'isdir', 'isfile', 'islink', 'ismount', 'join',
'normcase', 'normpath', 'os', 'split', 'splitdrive', 'splitext', 'splitunc',
'stat', 'string', 'varchars', 'walk']
```

Средства администрирования

Если этих огромных листингов недостаточно, чтобы двинуться дальше, поэкспериментируем с некоторыми из простейших инструментов os интерактивно. Как и sys, модуль os содержит набор средств для получения информации и администрирования:

```
>>> os.getpid()
-510737
>>> os.getcwd()
'C:\\PP2ndEd\\examples\\PP2E\\System'
>>> os.chdir(r'c:\temp')
>>> os.getcwd()
'c:\\temp'
```

Здесь показано, как функция os.getpid возвращает *ID процесса* (уникальный определяемый системой идентификатор выполняющейся программы), а os.getcwd возвраща-

ет *текущий рабочий каталог*. Текущим рабочим каталогом является тот, в котором предполагается нахождение файлов, открываемых вашим сценарием, если только в их имя не входит явный путь к каталогу. Вот почему ранее я говорил, чтобы вы выполнили именно в том каталоге, где находится *more.py*, следующую команду:

```
C:\...\PP2E\System>python more.py more.py
```

Аргумент с именем входного файла здесь задан без явного пути к каталогу (хотя его можно указать для постраничного вывода файлов из другого каталога). Если требуется выполнение в другом рабочем каталоге, вызовите функцию `os.chdir`, чтобы перейти в новый каталог: ваш код будет выполняться относительно нового каталога до конца программы (или нового вызова `os.chdir`). В этой главе еще будет сказано о понятии текущего рабочего каталога и его связи с импортом модулей, когда далее будет изучаться *контекст выполнения сценария*.

Константы переносимости

Модуль `os` экспортирует также ряд имен, упрощающих программирование для нескольких платформ. В этот набор входят специфические для платформ установки символов-разделителей для путей и каталогов, указателей родительского и текущего каталогов и символов, используемых для завершения строк на используемом компьютере:[1]

```
>>> os.pathsep, os.sep, os.pardir, os.curdir, os.linesep
(';', '\\', '..', '.', '\015\012')
```

Название `os.sep` используется для любого символа, используемого в качестве разделителя каталогов на той платформе, где выполняется Python; он будет автоматически установлен равным «\» для Windows, «/» для машин POSIX и «:» для Mac. Аналогично, `os.pathsep` задает символ, разделяющий каталоги в списках – «:» для POSIX и «;» для DOS и Windows. При использовании в сценариях таких атрибутов для составления и разбора относящихся к системе строк они становятся полностью переносимыми. Например, вызов вида `string.split(dirpath,os.sep)` правильно разберет на составляющие специфические для платформы имена каталогов, даже если `dirpath` выглядит как «dir\dir» в Windows, «dir/dir» в Linux и «dir:dir» на Macintosh.

Основные инструменты os.path

Вложенный модуль `os.path` предоставляет большой набор собственных средств, относящихся к каталогам. Например, в него входят переносимые функции для таких задач, как проверка типа файла (`isdir`, `isfile` и другие), подтверждение существования файла (`exists`), получение размера файла по его имени (`getsize`):

```
>>> os.path.isdir(r'C:\temp'),         os.path.isfile(r'C:\temp')
(1, 0)
>>> os.path.isdir(r'C:\config.sys'),   os.path.isfile(r'C:\config.sys')
(0, 1)
>>> os.path.isdir('nonesuch'),         os.path.isfile('nonesuch')
(0, 0)
>>> os.path.exists(r'c:\temp\data.txt')
0
>>> os.path.getsize(r'C:\autoexec.bat')
260
```

[1] `os.linesep` возвращается здесь как `\015\012` – восьмеричный эквивалент `\r\n`, отражая принятое в Windows соглашение: «возврат каретки + перевод строки». Смотрите обсуждение транслирования конца строки в разделе «Файловые инструменты» далее в этой главе.

Вызовы os.path.isdir и os.path.isfile говорят о том, является ли имя файла каталогом или простым файлом; оба они возвращают 0 («ложь»), если указанный файл не существует. Есть также вызовы для расщепления или соединения строк путей каталогов с использованием соглашений по именам каталогов той платформы, где работает Python:

```
>>> os.path.split(r'C:\temp\data.txt')
('C:\\temp', 'data.txt')
>>> os.path.join(r'C:\temp', 'output.txt')
'C:\\temp\\output.txt'

>>> name = r'C:\temp\data.txt'                          # пути Windows
>>> os.path.basename(name), os.path.dirname(name)
('data.txt', 'C:\\temp')

>>> name = '/home/lutz/temp/data.txt'                   # пути в стиле Unix
>>> os.path.basename(name), os.path.dirname(name)
('data.txt', '/home/lutz/temp')

>>> os.path.splitext(r'C:\PP2ndEd\examples\PP2E\PyDemos.pyw')
('C:\\PP2ndEd\\examples\\PP2E\\PyDemos', '.pyw')
```

Вызов os.path.split отделяет имя файла от пути к его каталогу, а os.path.join снова соединяет их вместе, и все совершенно переносимым образом с использованием соглашений по путям, действующих на той машине, на которой они вызываются. Вызовы basename и dirname просто возвращают здесь для удобства второй и первый элементы, возвращаемые split, а splitext отделяет расширение файла (за последней «.»). В этом модуле имеется также вызов abspath, который переносимым образом возвращает абсолютное полное имя каталога файла; он учитывает добавленный текущий каталог, родительский каталог «..» и другое:

```
>>> os.getcwd()
'C:\\PP2ndEd\\cdrom\\WindowsExt'
>>> os.path.abspath('temp')                     # расширит в полное имя пути
'C:\\PP2ndEd\\cdrom\\WindowsExt\\temp'
>>> os.path.abspath(r'..\examples')             # расширяются относительные пути
'C:\\PP2ndEd\\examples'
>>> os.path.abspath(r'C:\PP2ndEd\chapters')     # абсолютные пути не меняются
'C:\\PP2ndEd\\chapters'
>>> os.path.abspath(r'C:\temp\spam.txt')        # то же для имен файлов
'C:\\temp\\spam.txt'
>>> os.path.abspath('')                 # пустая строка означает текущий рабочий каталог (cwd)
'C:\\PP2ndEd\\cdrom\\WindowsExt'
```

Поскольку имена файлов считаются относящимися к текущему рабочему каталогу, если не заданы полными путями, функция os.path.abspath полезна, если нужно показать пользователю, какой каталог используется в действительности для сохранения файла. Под Windows, например, при запуске программ с графическим интерфейсом пользователя щелчком по значкам в проводнике или по ярлыкам на рабочем столе рабочим каталогом программы является тот, в котором находится щелкнутый файл, что не всегда очевидно пользователю: вывод abspath для файла может оказаться полезным.

Выполнение команд оболочки из сценариев

Модуль os позволяет также выполнять команды оболочки из сценариев Python. Это понятие переплетается с другими, которые будут освещаться в этой главе, но дальше,

однако ввиду важности этой идеи, используемой на протяжении всей данной части книги, бегло коснемся основ. Есть две функции os, позволяющие выполнить в сценарии любую командную строку, которую можно ввести в окне консоли:

`os.system`

Выполнить команду оболочки из сценария Python

`os.popen`

Выполнить команду оболочки из сценария и соединиться с ее входными или выходными потоками

Что такое команда оболочки?

Чтобы понять область действия этих вызовов, нужно сначала ввести несколько терминов. В данном тексте термин *оболочка* (*shell*) означает систему, которая считывает и выполняет на вашем компьютере командные строки, а *командой оболочки* (*shell command*) называется командная строка, которую вы обычно вводите в ответ на приглашение оболочки.

В Windows, например, можно открыть окно консоли MS-DOS и вводить в нем команды DOS – такие, как *dir* для получения списка каталогов, *type* для просмотра файла, имена программ, которые нужно запустить, и т. д. DOS является системной оболочкой, а команды типа *dir* и *type* являются командами оболочки. В Linux можно запустить новый сеанс оболочки, открыв окно xterm и тоже вводя в него команды оболочки – *ls* для перечисления каталогов, *cat* для просмотра файлов и т. д. Для Unix существует множество оболочек (например, csh, ksh), но все они читают и выполняют командные строки. Вот две команды оболочки, введенные и выполненные в окне консоли MS-DOS под Windows:

```
C:\temp>dir /B                    ... ввести командную строку оболочки
about-pp.html                     ... вот ее вывод
python1.5.tar.gz                  ... DOS служит оболочкой в Windows
about-pp2e.html
about-ppr2e.html
newdir

C:\temp>type helloshell.py
# программа Python
print 'The Meaning of Life'
```

Выполнение команд оболочки

Конечно, все это не имеет прямого отношения к Python (несмотря на то, что сценарии командной строки Python иногда ошибочно называют «инструментами оболочки» – shell tools). Но поскольку вызовы system и popen модуля os позволяют сценариям Python выполнять любые команды, понятные оболочке системы, можно использовать в своих сценариях любые имеющиеся на компьютере средства командной строки независимо от того, написаны они на Python или нет. Вот, например, код Python, который выполняет две команды оболочки DOS, введенные выше в ответ на приглашение оболочки:

```
C:\temp>python
>>> import os
>>> os.system('dir /B')
about-pp.html
python1.5.tar.gz
about-pp2e.html
about-ppr2e.html
```

```
newdir
0
>>> os.system('type helloshell.py')
# программа Python
print 'The Meaning of Life'
0
```

Выведенные в конце «0» являются значениями, возвращенными самим вызовом system. Вызов system можно использовать для выполнения любой командной строки, которую допускается ввести в ответ на приглашение оболочки (здесь им является C:\temp>). Выводимые командой данные обычно попадают в стандартный поток вывода сеанса Python или программы.

Обмен данными с командами оболочки

Но что если нужно перехватить в сценарии выдачу команды? Вызов os.system просто выполняет командную строку оболочки, но os.popen также соединяется со стандартными входным или выходным потоками команды – обратно возвращается объект типа файл, по умолчанию соединяемый с выходом команды (если передать popen флаг режима «w», то вместо этого произойдет соединение со входным потоком команды). Используя этот объект для чтения команды, запущенной с помощью popen, можно перехватывать текст, в обычных условиях появляющийся в окне консоли, в котором вводится команда:

```
>>> open('helloshell.py').read()
'# a Python program\012print 'The Meaning of Life'\012'

>>> text = os.popen('type helloshell.py').read()
>>> text
'# a Python program\012print 'The Meaning of Life'\012'

>>> listing = os.popen('dir /B').readlines()
>>> listing
['about-pp.html\012', 'python1.5.tar.gz\012', 'helloshell.py\012',
'about-pp2e.html\012', 'about-ppr2e.html\012', 'newdir\012']
```

Здесь мы получаем содержимое файла сначала обычным способом (средствами Python для работы с файлами), а затем как выдачу команды оболочки *type*. Чтение выдачи команды *dir* позволяет получить список файлов в каталоге, который можно затем обработать в цикле (далее в этой главе мы познакомимся с другими способами получения такого списка). До сих пор мы выполняли базовые команды DOS, но поскольку эти вызовы могут выполнить *любую* командную строку, вводимую в ответ на системное приглашение, с их помощью можно также запускать другие сценарии Python:

```
>>> os.system('python helloshell.py')             # запуск программы Python
The Meaning of Life
0
>>> output = os.popen('python helloshell.py').read()
>>> output
'The Meaning of Life\012'
```

Во всех этих примерах командные строки, передаваемые system и popen, жестко закодированы, но нет причин, по которым программы Python не могли бы *создавать* такие строки на этапе исполнения с помощью обычных строковых операций (+, % и т. д.). При условии, что команды могут динамически строиться и выполняться, system и popen превращают сценарии Python в гибкие и переносимые средства для запуска и управления другими программами. Например, тестовый «управляющий» сценарий Python можно использовать для запуска программ, написанных на любых языках

(например, C++, Java, Python), и анализа их выдачи. Такой сценарий будет рассмотрен в разделе «Сценарий регрессивного испытания» главы 4 «Более крупные системные примеры, часть 2».

Ограничения, присущие командам оболочки

Следует помнить о двух ограничениях, связанных с `system` и `popen`. Во-первых, хотя сами по себе эти функции хорошо переносимы, в действительности их применение переносимо лишь в той мере, в какой это относится к выполняемым ими командам. Например, предыдущие примеры с командами DOS *dir* и *type* работают только в Windows, а для платформ Unix они должны быть переработаны и выполнять команды *ls* и *cat*. На момент написания этих строк вызов `popen` в Windows работал только в программах командной строки: он оказывался неуспешным при вызове из программы, выполняющейся в Windows с какого-либо рода пользовательским интерфейсом (например, под GUI IDLE для разработок Python). В версии Python 2.0 это было исправлено – `popen` теперь гораздо лучше работает под Windows, – но для такого исправления, естественно, необходима установка на машину самой последней версии Python.

Во-вторых, важно помнить, что запуск таким способом файлов Python как программ очень отличается от импорта программных файлов и вызова определенных в них функций и обычно происходит гораздо медленнее. Когда вызываются `os.system` и `os.popen`, им приходится запускать совершенно новую и независимую программу, выполняемую операционной системой (на Unix-подобных платформах команды выполняются в новых ответвленных процессах). При *импорте* программы в качестве модуля интерпретатор Python просто загружает и выполняет код файла в том же процессе, генерируя объект модуля. При этом другие программы не запускаются.[1]

Могут быть веские основания для построения системы в виде отдельных программ, и далее будут рассмотрены такие темы, как аргументы командной строки и потоки, которые позволяют программам передавать информацию в обоих направлениях. Но в большинстве случаев для построения систем более быстрым и прямым способом является импорт модулей.

Если вы серьезно намерены использовать эти вызовы, то следует знать, что вызов `os.system` обычно *блокирует* (то есть приостанавливает) вызвавшую его программу до завершения выполнения порожденной команды. В Linux и Unix-подобных платформах порожденную команду обычно можно заставить выполняться независимо и параллельно с вызвавшей ее программой, добавив в конец командной строки оператор фонового выполнения `&`:

```
os.system("python program.py arg arg &")
```

В Windows запуск с помощью команды DOS *start* обычно также влечет параллельное выполнение команды:

```
os.system("start program.py arg arg")
```

Вызов `os.popen` обычно не блокирует вызвавшего, он, по определению, должен иметь возможность читать или писать в возвращаемый файловый объект. Но вызвавший

---

[1] Встроенная функция Python `execfile` тоже выполняет код программного файла, но внутри того же процесса, который ее вызвал. В этом отношении она аналогична импорту, но работа ее больше похожа на то, как если бы текст файла был *вставлен* в вызывающую программу в том месте, где стоит вызов `execfile` (если не передаются явные словари глобальных или локальных пространств имен). В отличие от импорта, `execfile` безусловно читает и выполняет код файла (он может выполняться несколько раз в одном процессе), и при выполнении файла не создается объект модуля.

все же может иногда оказаться заблокированным – как в Windows, так и в Linux, – если объект канала закрыт до завершения порожденной программы (например, при сборке мусора), или канал считывается до исчерпания (например, с помощью метода канала `read()`). Как будет показано в следующей главе, для параллельного исполнения программ без блокирования можно использовать вызов `os.fork/exec` в Unix и `os.spawnv` – в Windows.

Поскольку вызовы `os system` и `popen` попадают также в категорию средств запуска программ, переадресации потоков и взаимодействия между процессами, они снова появятся далее в этой и последующих главах, поэтому дальнейшие детали мы пока отложим.

Другие элементы, экспортируемые модулем os

Поскольку все другие инструменты модуля `os` еще труднее оценить вне контекста более крупных приложений, мы отложим их более пристальное рассмотрение до последующих разделов. Но чтобы дать вам представление о характере этого модуля, ниже приведен краткий справочный обзор. Среди прочего на вооружении модуля `os` состоят:

`os.environ`

Получение и установка переменных окружения оболочки.

`os.fork`

Порождение новых дочерних процессов в Unix.

`os.pipe`

Обмен данными между программами.

`os.execlp`

Запуск новых программ.

`os.spawnv`

Запуск новых программ под Windows.

`os.open`

Открытие файла с использованием файлового дескриптора.

`os.mkdir`

Создание каталога.

`os.mkfifo`

Создание нового именованного канала.

`os.stat`

Получение информации низкого уровня о файле.

`os.remove`

Удаление файла с использованием пути к нему.

`os.path.walk`

Применение функции к файлам в целом дереве каталогов.

И так далее. Заранее предупреждаем: модуль `os` предоставляет группу вызовов для открытия, чтения и записи файлов, но все они используют доступ к файлам на низком уровне и совершенно отличаются от встроенных файловых объектов Python `stdio`, создаваемых с помощью встроенной функции `open`. Обычно во всех случаях следует использовать встроенную функцию `open` (а не модуль `os`), кроме очень специфических потребностей обработки файлов.

Повсюду в этой главе для решения обычных системных задач будут использоваться такого рода инструменты из sys и os, но объем данной книги не позволяет приводить полные списки содержимого встречающихся модулей. Если вы этого еще не сделали, следует знакомиться с содержимым таких модулей, как os и sys, обратившись к руководству по библиотеке Python. А теперь перейдем к исследованию дополнительных системных инструментов в контексте более широких понятий системного программирования.

Контекст выполнения сценария

Сценарии Python выполняются не в вакууме. В зависимости от платформы и процедуры начального запуска программы Python могут находиться в любом виде охватывающего *контекста* – информации, автоматически передаваемой программе операционной системой при запуске программы. Например, у сценариев есть доступ к следующим видам входных данных системного уровня и интерфейсов:

Текущий рабочий каталог

os.getcwd предоставляет доступ к каталогу, из которого запущен сценарий и значение которого неявно используют многие средства работы с файлами.

Аргументы командной строки

sys.argv предоставляет доступ к словам, введенным в командной строке, с помощью которой была запущена программа, и служащим в качестве входных данных сценария.

Переменные оболочки

os.environ предоставляет интерфейс к именам, назначаемым в охватывающей оболочке (или родительской программе) и передаваемым сценарию.

Стандартные потоки

sys.stdin, stdout и stderr экспортируют три потока ввода/вывода, лежащие в центре инструментов оболочки командной строки.

Такие средства могут служить в качестве входных данных сценариев, параметров конфигурации и т. д. Несколько последующих разделов исследуют эти инструменты контекста – их интерфейсы к Python и обычно выполняемую роль.

Текущий рабочий каталог

Понятие текущего рабочего каталога (CWD, current working directory) оказывается ключевым при выполнении некоторых сценариев: это всегда неявно определенное место, где предполагается размещение обрабатываемых сценарием файлов, если их имена не содержат абсолютных путей каталогов. Как уже указывалось выше, os.getcwd позволяет сценарию получить имя CWD в явном виде, а os.chdir позволяет сценарию переместиться в новый CWD.

Имейте, однако, в виду, что для имен файлов, не содержащих полного пути, подразумевается, что они находятся в CWD, и это не имеет никакого отношения к установке PYTHONPATH. Технически, CWD всегда указывает на то место, откуда запущен сценарий, а не на каталог, содержащий файл сценария. Напротив, при *импорте* поиск всегда осуществляется в каталоге, содержащем сценарий, а не в CWD (если только сценарий тоже не размещен в CWD). Поскольку это тонкое различие, на котором часто попадаются новички, изучим его более подробно.

CWD, файлы и пути для импорта

Если запустить сценарий Python, введя в оболочке командную строку типа `python dir1\dir2\file.py`, то CWD будет представлять тот каталог, в котором вы находились, вводя эту команду, но не *dir1\dir2*. С другой стороны, Python автоматически помещает путь к исходному каталогу сценария перед путем поиска модулей, поэтому *file.py* всегда может импортировать другие файлы из *dir1\dir2*, откуда бы он ни был запущен. Чтобы проиллюстрировать это, напишем простой сценарий, выводящий CWD и путь поиска модулей:

```
C:\PP2ndEd\examples\PP2E\System>type whereami.py
import os, sys
print 'my os.getcwd =>', os.getcwd()            # показать каталог исполнения cwd
print 'my sys.path  =>', sys.path[:6]           # показать первые 6 путей для импорта
raw_input()                                     # ждать нажатия клавиши
```

Теперь при запуске этого сценария в том каталоге, где он находится, CWD устанавливается ожидаемым образом, а перед путем поиска модулей добавляется пустая строка (`''`), обозначающая CWD (путь поиска модулей `sys.path` нам уже встречался):

```
C:\PP2ndEd\examples\PP2E\System>set PYTHONPATH=C:\PP2ndEd\examples
C:\PP2ndEd\examples\PP2E\System>python whereami.py
my os.getcwd => C:\PP2ndEd\examples\PP2E\System
my sys.path  => ['', 'C:\\PP2ndEd\\examples', 'C:\\Program Files\\Python
\\Lib\\plat-win', 'C:\\Program Files\\Python\\Lib', 'C:\\Program Files\\
Python\\DLLs', 'C:\\Program Files\\Python\\Lib\\lib-tk']
```

Но если запускать этот сценарий из других мест, туда же будет перемещаться CWD (это каталог, в котором вводятся команды), и Python будет добавлять в начало пути поиска модулей каталог, который позволяет сценарию по-прежнему видеть файлы своего исходного каталога. Например, если запустить сценарий, поднявшись на один уровень (`'..'`), первым каталогом, в котором Python станет искать импортируемые модули из *whereami.py*, станет имя «System», добавленное в начало `sys.path`: оно нацеливает импорт обратно на каталог, содержащий выполняемый сценарий. Однако имена файлов, не содержащие полного пути, будут отображаться в CWD (*C:\PP2ndEd\examples\PP2E*), а не в его подкаталог *System*:

```
C:\PP2ndEd\examples\PP2E\System>cd ..
C:\PP2ndEd\examples\PP2E>python System\whereami.py
my os.getcwd => C:\PP2ndEd\examples\PP2E
my sys.path  => ['System', 'C:\\PP2ndEd\\examples',          ...остальное так же... ]
C:\PP2ndEd\examples\PP2E>cd ..
C:\PP2ndEd\examples>python PP2E\System\whereami.py
my os.getcwd => C:\PP2ndEd\examples
my sys.path  => ['PP2E\\System', 'C:\\PP2ndEd\\examples',    ...остальное так же... ]
C:\PP2ndEd\examples\PP2E\System>cd PP2E\System\App
C:\PP2ndEd\examples\PP2E\System\App>python ..\whereami.py
my os.getcwd => C:\PP2ndEd\examples\PP2E\System\App
my sys.path  => ['..', 'C:\\PP2ndEd\\examples',              ...остальное так же... ]
```

В результате *имена файлов* в сценарии, не содержащие путей каталогов, будут искаться в том месте, где была введена команда (`os.getcwd`), но *импорт* по-прежнему сохранит доступ к каталогу выполняемого сценария (через начало `sys.path`). Наконец, если файл запускается щелчком по значку, CWD является просто каталогом, содержащим файл, по которому сделан щелчок. Например, следующая выдача появляется в новом окне консоли DOS при двойном щелчке по *whereami.py* в проводнике Windows:

```
my os.getcwd => C:\PP2ndEd\examples\PP2E\System
```

```
my sys.path  => ['C:\\PP2NDED\\EXAMPLES\\PP2E\\SYSTEM', 'C:\\PP2ndEd\\examples',
'C:\\Program Files\\Python\\Lib\\plat-win', 'C:\\Program Files\\Python\\Lib',
'C:\\Program Files\\Python\\DLLs']
```

В данном случае CWD, используемый для имен файлов, и первый каталог для поиска импорта являются каталогом, содержащим файл сценария. Обычно это действует так, как предполагается, но нужно избегать двух ловушек:

- Имена файлов должны содержать полные пути каталогов, если сценарии не знают точно, из какого места они запущены.
- Сценарии командной строки не могут использовать CWD, чтобы видеть импортируемые файлы, не находящиеся в их собственных каталогах; для доступа к модулям из других каталогов нужно использовать установки PYTHONPATH и пути импорта пакетов.

Например, файлы из этой книги всегда могут импортировать другие файлы из собственного исходного каталога без указания пути пакета для импорта, независимо от того, как они запущены (`import filehere`), но должны пройти через корень пакета *PP2E*, чтобы найти файлы в другом месте дерева примеров (`from PP2E.dir1.dir2 import filethere`), даже если запустить их из каталога, содержащего нужный внешний модуль. Как обычно для модулей, имя каталога *PP2E\dir1\dir2* можно также добавить в PYTHONPATH, чтобы сделать `filethere` видимым всюду без указания пути пакета для импорта (хотя лишние каталоги в PYTHONPATH увеличивают вероятность конфликта имен). Однако в любом случае импорт всегда осуществляется из исходного каталога сценария или другого пути поиска в установках Python, а не из CWD.

Текущий рабочий каталог и командные строки

Это различие между CWD и путями поиска для импорта объясняет, почему многие сценарии в данной книге, которые должны действовать в текущем рабочем каталоге (а не в том, имя которого передано), запускаются командной строкой вида:

```
C:\temp>python %X%\PyTools\cleanpyc-py.py                    обработка cwd
```

В данном примере сам файл сценария Python находится в каталоге *C:\PP2ndEd\examples\PP2E\PyTools*, но поскольку он запускается из *C:\temp*, то обрабатывает файлы, содержащиеся в *C:\temp* (то есть в CWD, а не исходном каталоге сценария). Чтобы обработать с помощью такого сценария файлы, находящиеся где-то в другом месте, нужно просто изменить CWD с помощью *cd* и перейти в каталог, который должен быть обработан:

```
C:\temp>cd C:\PP2nEd\examples
C:\PP2ndEd\examples>python %X%\PyTools\cleanpyc-py.py      обработка cwd
```

Поскольку CWD всегда неявно определен, то *cd* сообщает сценарию, какой каталог должен быть обработан, не менее определенно, чем при явной передаче имени каталога сценарию, например, в виде:

```
C:\...\PP2E\PyTools>python find.py *.py C:\temp    обработка указанного каталога
```

В этой командной строке CWD является каталогом, содержащим имя сценария, который должен быть выполнен (обратите внимание на отсутствие префикса пути в имени файла сценария); но поскольку этот сценарий обрабатывает каталог, явно указанный в командной строке (*C:\temp*), CWD несуществен. Наконец, если нужно выполнить такой сценарий, расположенный в некотором другом каталоге, для обработки файлов, находящихся в третьем каталоге, можно просто указать пути каталогов для обоих:

```
C:\temp>python %X%\PyTools\find.py *.cxx C:\PP2ndEd\examples\PP2E
```

В этом случае видимыми для импорта являются файлы в исходном каталоге сценария *PP2E\PyTools*, и обрабатываются файлы корневого каталога *PP2E*, но CWD представляет совершенно другой каталог (*C:\temp*). При этом последнем формате приходится, конечно, больше вводить с клавиатуры, но в этой книге вы встретитесь с разнообразными CWD и командными строками с явными путями к сценариям.

Если вы встречаете в командной строке `%X%`, как в предшествующих примерах, то знайте, что это ссылка на значение переменной окружения оболочки с именем `X`. Это всего лишь сокращение для полного пути к каталогу *PP2E* – корневому для примеров в книге, которое я использую для указания на файлы сценариев. На моих машинах она устанавливается в файлах `PP2E\Config setup-pp*` следующим образом:

```
set X=C:\PP2ndEd\examples\PP2E            --DOS
setenv X /home/mark/PP2ndEd/examples/PP2E --Unix/csh
```

Это означает, что она получает значение и расширяется в каталог, в котором на машине находится *PP2E*. Подробности смотрите в файлах `Config\setup-pp*`, а также читайте далее в этой главе о переменных оболочки. Вместо `%X%` в любом месте этой книги можно вводить полные пути, но для пальцев и клавиатуры будет, вероятно, лучше присвоить `X` значение корневого каталога с примерами.

Аргументы командной строки

Модуль `sys` позволяет также получить от Python те слова, которые были введены в команде, запустившей сценарий Python. Эти слова обычно называются аргументами командной строки и находятся во встроенном списке строк `sys.argv`. Программисты на C могут обнаружить сходство с массивом C «argv» (массивом строк C). В интерактивном режиме смотреть особенно не на что, так как для запуска Python в этом режиме в командной строке аргументов не передается:

```
>>> sys.argv
['']
```

Чтобы действительно увидеть аргументы, нужно запустить сценарий из командной строки. В примере 2.2 показан безобразно простой сценарий, который всего лишь выводит список `argv` для изучения.

Пример 2.2. PP2E\System\testargv.py

```
import sys
print sys.argv
```

При выполнении этого сценария выводится список аргументов командной строки. Обратите внимание, что первым элементом всегда является имя самого выполняемого сценария Python независимо от способа запуска сценария (см. врезку «Выполняемые сценарии в Unix» далее в этой главе):

```
C:\...\PP2E\System>python testargv.py
['testargv.py']

C:\...\PP2E\System>python testargv.py spam eggs cheese
['testargv.py', 'spam', 'eggs', 'cheese']

C:\...\PP2E\System>python testargv.py -i data.txt -o results.txt
['testargv.py', '-i', 'data.txt', '-o', 'results.txt']
```

Последняя команда иллюстрирует общепринятое соглашение. Подобно аргументам функции, параметры командной строки иногда передаются по позиции, а иногда по имени с помощью пары «-имя значение». Например, пара `-i data.txt` означает, что значением опции `-i` является `data.txt` (например, имя файла). Перечислять можно любые слова, но обычно программы накладывают на них некоторые структурные ограничения.

Аргументы командной строки играют в программах такую же роль, как аргументы функций в функциях: они просто позволяют передать в программу информацию, которая может быть различной для каждого запуска программы. То, что они не кодируются жестким образом, позволяет использовать сценарии более общим образом. Например, сценарий для обработки файлов может использовать аргумент командной строки для передачи имени файла, который должен быть им обработан, – взгляните на сценарий *more.py*, который был нашим первым примером. Другие сценарии могут принимать флаги режима обработки, адреса Интернета и т. д.

Однако при регулярном использовании аргументов командной строки вы можете обнаружить, что писать код, который вылавливает в списке слова, неудобно. Обычно программы переводят при запуске список аргументов в структуры, которые обрабатывать удобнее. Вот один из способов сделать это: сценарий в примере 2.3 просматривает список `argv` в поисках пар слов `-optionname optionvalue` и размещает их в словаре по именам опций, чтобы потом их было легче извлечь.

Пример 2.3. PP2E\System\testargv2.py

```
# собрать опции командной строки в словаре

def getopts(argv):
    opts = {}
    while argv:
        if argv[0][0] == '-':                   # найти пары "-имя значение"
            opts[argv[0]] = argv[1]             # ключ словаря - аргумент "-name"
            argv = argv[2:]
        else:
            argv = argv[1:]
    return opts

if ____name____ == '__main__':
    from sys import argv                        # пример кода клиента
    myargs = getopts(argv)
    if myargs.has_key('-i'):
        print myargs['-i']
    print myargs
```

Такую функцию можно импортировать и использовать во всех инструментах командной строки. При самостоятельном выполнении этот файл просто выводит форматированный словарь аргументов:

```
C:\...\PP2E\System>python testargv2.py
{}

C:\...\PP2E\System>python testargv2.py -i data.txt -o results.txt
data.txt
{'-o': 'results.txt', '-i': 'data.txt'}
```

Естественно, можно гораздо дальше углубиться здесь в сложности схем аргументов, проверки ошибок и тому подобного. Можно также использовать стандартные и более развитые средства обработки командной строки библиотеки Python для анализа аргументов; в качестве другого варианта посмотрите модуль `getopt` в руководстве по биб-

лиотеке. В целом, чем более доступны для настройки ваши сценарии, тем большая сложность закладывается в логику обработки командной строки.

Переменные окружения оболочки

Переменные оболочки, которые иногда называют также переменными окружения, доступны в сценариях Python через `os.environ` – объект Python типа словаря, в котором есть по одной записи для каждой переменной, установленной в оболочке. Переменные оболочки располагаются вне Python: они часто устанавливаются в командной строке или в файлах начального запуска и обычно служат в качестве общесистемных настроечных данных для программ.

На самом деле вам уже должен быть знаком главный пример: путь поиска модулей PYTHONPATH является переменной оболочки, которую Python использует при импорте модулей. После установки ее значения в файлах начального запуска системы оно становится доступным для каждой запускаемой программы Python. Переменные окружения могут также устанавливаться программами, чтобы передавать входные данные другим программам в приложении; поскольку обычно их значения наследуются порожденными программами, они могут служить в качестве простого средства связи между процессами.

Выполняемые файлы в Unix

Пользователям Unix и Linux: текстовые файлы с исходным кодом Python можно сделать непосредственно исполняемыми, добавив в их начало особую строку, содержащую путь к интерпретатору Python, и присвоив файлу права на выполнение. Например, введите в текстовый файл с именем «myscript» следующий код:

```
#!/usr/bin/python
print 'And nice red uniforms'
```

Первая строка должна восприниматься Python как комментарий (она начинается с `#`), но при запуске этого файла операционная система посылает строки этого файла интерпретатору, указанному после `#!` в строке 1. Если этот файл сделать непосредственно исполняемым с помощью команды оболочки вида `chmod +x myscript`, его можно выполнять непосредственно, не вводя в команде `python`, как если бы это был двоичный файл исполняемой программы:

```
% myscript a b c
And nice red uniforms
```

При запуске таким способом `sys.argv` по-прежнему будет содержать имя сценария в качестве первого слова в списке: [«myscript», «a», «b», «c»] – в точности, как если бы сценарий был запущен с помощью более явного и переносимого формата команды `python myscript a b c`. Превращение сценариев в непосредственно исполнимые файлы в действительности является трюком Unix, а не функцией Python, но стоит отметить, что можно сделать его несколько менее машинно-зависимым, указав в начале команду Unix *env* вместо жестко зашитого пути к исполняемому файлу Python:

```
#!/usr/bin/env python
print 'Wait for it...'
```

При кодировании таким способом операционная система находит интерпретатор Python с помощью значений переменных окружения (обычно переменной PATH). Выполняя один и тот же сценарий на разных машинах, нужно только изменять на них настройки окружения, а не редактировать код сценария Python. Конечно, файлы Python по-прежнему можно запускать с помощью более явной командной строки:

```
% python myscript a b c
```

При этом предполагается, что интерпретирующая программа `python` находится в системном пути поиска (иначе нужно указывать полный путь к ней), но это действует на любой платформе с Python и командной строкой. Поскольку это более переносимый способ, я обычно использую его в примерах книги, однако посмотрите на своих страницах руководства Unix дополнительные сведения по упоминавшимся здесь темам. Несмотря на это, особые строчки `#!` можно встретить во многих примерах данной книги – на случай, если читателям потребуется запускать их как исполняемые файлы в Unix или Linux; на других платформах они просто игнорируются как комментарии Python. Заметим, что в Windows NT/2000 обычно можно прямо ввести имя сценария (без слова «python»), чтобы запустить его, и добавлять в начале строку `#!` не нужно.

Получение значений переменных оболочки

В Python окружение оболочки является простым предустановленным объектом без специального синтаксиса. Задание в качестве индекса для `os.environ` строки с именем нужной переменной оболочки (например, `os.environ['USER']`) является моральным эквивалентом знака доллара перед именем переменной для большинства оболочек Unix (например, `$USER`), использования с двух сторон процента в DOS (`%USER%`) и вызова `getenv("USER")` в программе C. Начнем интерактивный сеанс и поэкспериментируем:

```
>>> import os
>>> os.environ.keys()
['WINBOOTDIR', 'PATH', 'USER', 'PP2HOME', 'CMDLINE', 'PYTHONPATH', 'BLASTER',
'X', 'TEMP', 'COMSPEC', 'PROMPT', 'WINDIR', 'TMP']
>>> os.environ['TEMP']
'C:\\windows\\TEMP'
```

Здесь метод `keys` возвращает список установленных переменных, а с помощью индекса получается значение переменной TEMP в Windows. В Linux это действует таким же образом, но обычно при запуске Python установлены другие переменные. Поскольку нам знакома переменная PYTHONPATH, посмотрим в Python на ее значение и убедимся в его правильности:[1]

```
>>> os.environ['PYTHONPATH']
'C:\\PP2ndEd\\examples\\Part3;C:\\PP2ndEd\\examples\\Part2;C:\\PP2ndEd\\
examples\\Part2\\Gui;C:\\PP2ndEd\\examples'
>>>
>>> import string
>>> for dir in string.split(os.environ['PYTHONPATH'], os.pathsep):
...   print dir
```

[1] Эти результаты отражают старую установку пути, использовавшуюся во время разработки; сейчас эта переменная содержит лишь один каталог, в котором находится корень *PP2E*.

```
...
C:\PP2ndEd\examples\Part3
C:\PP2ndEd\examples\Part2
C:\PP2ndEd\examples\Part2\Gui
C:\PP2ndEd\examples
```

PYTHONPATH является строкой, состоящей из каталогов, разделяемых символом, который используется для разделения таких элементов на вашей платформе (например, «;» в DOS/Windows, «:» в Unix и Linux). Чтобы разделить эту строку на составляющие, в `string.split` передается разделитель `os.pathsep` – переносимая установка, дающая правильный разделитель для соответствующей машины.

Изменение переменных оболочки

Как и обычные словари, объект `os.environ` поддерживает обращение по ключу и *присваивание*. Как обычно, присваивание изменяет значение ключа:

```
>>> os.environ['TEMP'] = r'c:\temp'
>>> os.environ['TEMP']
'c:\\temp'
```

Но при этом происходит нечто еще. В последних версиях Python значения, присваиваемые таким образом `os.environ`, автоматически *экспортируются* в другие части приложения. Это означает, что присваивание ключу изменяет объект `os.environ` в программе Python и соответствующую переменную в окружении *оболочки* для процесса выполняемой программы. Ее новое значение становится видимым программе Python, всем связанным с ней модулям на C и всем программам, порождаемым процессом Python. Внутренне при присваивании по ключу `os.environ` происходит вызов `os.putenv` – функции, изменяющей переменную окружения за границами интерпретатора Python. Чтобы показать, как это работает, нужна пара сценариев, устанавливающих переменные оболочки и получающих их значения. Первый из них показан в примере 2.4.

Пример 2.4. PP2E\System\Environment\setenv.py

```
import os
print 'setenv...',
print os.environ['USER']               # показать текущее значение переменной оболочки

os.environ['USER'] = 'Brian'           # неявно выполняет os.putenv
os.system('python echoenv.py')

os.environ['USER'] = 'Arthur'          # изменение передается порожденным программам
os.system('python echoenv.py')         # и связанным с процессом модулям на C

os.environ['USER'] = raw_input('?')
print os.popen('python echoenv.py').read()
```

Данный сценарий *setenv.py* просто изменяет переменную оболочки USER и запускает другой сценарий, выводящий значение этой переменной, который показан в примере 2.5.

Пример 2.5. PP2E\System\Environment\echoenv.py

```
import os
print 'echoenv...',
print 'Hello,', os.environ['USER']
```

Независимо от способа запуска, *echoenv.py* выводит значение USER в окружающей оболочке; при запуске из командной строки это значение является тем, которое мы установили в самой оболочке:

```
C:\...\PP2E\System\Environment>set USER=Bob
C:\...\PP2E\System\Environment>python echoenv.py
echoenv... Hello, Bob
```

Однако при запуске из другого сценария, например *setenv.py*, сценарий *echoenv.py* получает то значение USER, которое установила его родительская программа:

```
C:\...\PP2E\System\Environment>python setenv.py
setenv... Bob
echoenv... Hello, Brian
echoenv... Hello, Arthur
?Gumby
echoenv... Hello, Gumby
C:\...\PP2E\System\Environment>echo %USER%
Bob
```

Точно так же это действует в Linux. В целом порожденная программа всегда *наследует* значения переменных окружения своих родителей. «Порожденными» программами являются такие, которые запускаются средствами Python, например, `os.spawnv` в Windows, комбинацией `os.fork/exec` в Unix и Linux и `os.popen` и `os.system` на ряде других платформ, – все программы, запущенные таким образом, получают установки переменных окружения, существующие в момент запуска у родительского процесса.[1]

Подобная установка переменных окружения перед запуском новой программы является одним из способов передачи информации в новую программу. Например, сценарий конфигурирования Python может настроить переменную PYTHONPATH так, чтобы она включала пользовательские каталоги, прежде чем запускать другой сценарий Python; у запущенного сценария будет свой путь для поиска, потому что переменные оболочки передаются потомкам (такой запускающий сценарий будет приведен в конце главы 4).

Обратите, однако, внимание на последнюю программу предыдущего сценария: переменная USER получает свое первоначальное значение после завершения программы Python верхнего уровня. Присвоения значений ключам `os.environ` передаются из интерпретатора *вниз* по цепочке порожденных программ и никогда не передаются *вверх* процессам родительских программ (включая системную оболочку). Это относится и к C-программам, использующим библиотечный вызов `putenv`, не являясь ограничением собственно Python. Это едва ли вызовет проблемы в сценарии Python, находящемся в вершине приложения. Но помните, что настройки оболочки, сделанные внутри программы, действуют, лишь пока выполняется эта программа и порожденные ею дочерние программы.

[1] Так происходит по умолчанию. Некоторые средства запуска программ позволяют сценариям передавать в дочерние программы значения переменных окружения, отличные от своих собственных. Например, вызов `os.spawnve` аналогичен `os.spawnv`, но принимает аргумент типа словаря, представляющий окружение оболочки, которое должно быть передано запускаемой программе. Некоторые разновидности `os.exec*` (имена которых оканчиваются на «e») тоже принимают явное задание окружения; детали формата вызова `os.exec` смотрите в главе 3 «Системные средства параллельного выполнения».

Стандартные потоки

Модуль sys – это место обитания стандартных потоков ввода, вывода и ошибок для программ Python:

```
>>> for f in (sys.stdin, sys.stdout, sys.stderr): print f
...
<open file '<stdin>', mode 'r' at 762210>
<open file '<stdout>', mode 'w' at 762270>
<open file '<stderr>', mode 'w' at 7622d0>
```

Стандартные потоки просто являются заранее открытыми файловыми объектами Python, автоматически соединяемыми со стандартными потоками программы при запуске Python. По умолчанию все они связаны с окном консоли, в котором был запущен Python (или программа Python). Поскольку утверждение print и функция raw_input являются ничем иным, как дружественными пользователю интерфейсами к стандартным потокам вывода и ввода, они аналогичны прямому использованию stdout и stdin в sys:

```
>>> print 'hello stdout world'
hello stdout world
>>> sys.stdout.write('hello stdout world' + '\n')
hello stdout world
>>> raw_input('hello stdin world>')
hello stdin world>spam
'spam'
>>> print 'hello stdin world>',; sys.stdin.readline()[:-1]
hello stdin world>eggs
'eggs'
```

Стандартные потоки в Windows

Пользователям Windows: если в проводнике Windows для запуска программы Python щелкнуть по имени ее файла с расширением *.py* (или запустить ее через os.system), автоматически появляется окно консоли DOS, служащее стандартным потоком программы. Если программа создает собственные окна, можно избежать открытия окна консоли, дав файлу с исходным кодом программы расширение *.pyw*, а не *.py*. Расширение *.pyw* означает просто исходный файл *.py* без всплывающего в Windows окна DOS.

Предупреждение: в версии Python 1.5.2 файлы *.pyw* можно только выполнять, но не импортировать: *.pyw* не воспринимается как имя модуля. Если требуется, чтобы программа одновременно выполнялась без всплывающей консоли DOS и могла импортироваться, нужны оба файла – *.py* и *.pyw*. Файл *.pyw* может быть просто сценарием верхнего уровня, импортирующим и вызывающим основную логику программы, размещенную в *.py*. Пример этого есть в главе 9 «Более крупные примеры GUI» в разделе «PyEdit: объект и программа текстового редактора/объект».

Заметьте также, что поскольку при щелчке по программе вывод производится в это всплывающее окно DOS, сценарии, просто выводящие текст и завершающие свою работу, производят странную «вспышку»: всплывает окно консоли DOS, в него производится вывод, а затем окно сразу закрывается (не самое дружественное пользователю поведение!). Чтобы сохранить окно DOS и иметь возможность прочесть в нем выдачу, добавьте просто вызов raw_input() в конец сценария, что создаст остановку перед нажатием клавиши Enter для выхода.

Переадресация потоков в файлы и программы

Формально текст стандартного вывода (и функции `print`) появляется в окне консоли, в котором запущена программа, текст стандартного ввода (и `raw_input`) поступает с клавиатуры, а стандартный вывод ошибок используется для вывода сообщений Python об ошибках в окно консоли. По крайней мере, так происходит по умолчанию. Существует также возможность переадресации этих потоков в файлы или в другие программы системной оболочки, а также произвольным объектам внутри сценария Python. В большинстве систем с помощью такой переадресации облегчается повторное использование и комбинирование утилит командной строки общего назначения.

Переадресация потоков в файлы

Переадресация удобна для таких вещей, как готовый (предварительно созданный) ввод для тестирования: один и тот же тестовый сценарий можно применять с некоторым набором входных данных, просто переадресуя при каждом запуске сценария стандартный входной поток на разные файлы. Аналогично переадресация стандартного потока вывода позволяет сохранить и впоследствии проанализировать вывод программы; например, в тестирующих системах для обнаружения ошибок может сравниваться сохраненный стандартный вывод сценария с файлом, содержащим ожидаемую выдачу.

Несмотря на всю мощь этой парадигмы, использовать переадресацию весьма просто. Рассмотрим, например, простой программный цикл «прочесть-вычислить-вывести» в примере 2.6.[1]

Пример 2.6. PP2E\System\Streams\teststreams.py

```
# считывать числа до символа конца файла (eof) и выводить их квадраты

def interact():
    print 'Hello stream world'                        # print выводит на sys.stdout
    while 1:
        try:
            reply  = raw_input('Enter a number>')     # raw_input читает из sys.stdin
        except EOFError:
           break                                  # возбуждает исключительную ситуацию при eof
        else:                                         # входные данные в виде строки
            num = int(reply)
            print "%d squared is %d" % (num, num ** 2)
    print 'Bye'

if __name__ == '__main__':
    interact()                                        # если выполняется, а не импортируется
```

Как обычно, функция `interact` вызывается здесь автоматически, если файл выполняется, а не импортируется. По умолчанию запуск этого файла из командной строки системы вызывает появление стандартного потока при вводе команды Python. Сцена-

[1] Обратите внимание, что `raw_input` возбуждает исключительную ситуацию, сигнализируя о конце файла, однако методы чтения файлов просто возвращают в этом случае пустую строку. Поскольку `raw_input` также обрезает символ конца строки в концах строк, то пустая строка в файле тоже возвращает пустую строку, поэтому исключительная ситуация необходима для указания состояния конца файла. Методы чтения файлов сохраняют символ завершения строки и возвращают для пустых строк `\n` , а не `""`. Это одно из различий между прямым чтением `sys.stdin` и `raw_input`. Последняя также принимает в качестве аргумента строку приглашения, автоматически выводимую перед приемом входных данных.

рий просто читает числа, пока не достигнет конца файла в стандартном входном потоке (в Windows конец файла обычно задается комбинацией двух клавиш <Ctrl>+<Z>; в Unix нужно ввести <Ctrl>+<D>):

```
C:\...\PP2E\System\Streams>python teststreams.py
Hello stream world
Enter a number>12
12 squared is 144
Enter a number>10
10 squared is 100
Enter a number>
```

Как в Windows, так и на Unix-подобных платформах, стандартный входной поток можно переадресовать так, чтобы он поступал из файла, с помощью синтаксиса оболочки < *filename*. Вот командный сеанс в окне консоли DOS под Windows, который вынуждает сценарий читать входные данные из текстового файла *input.txt*. Под Linux происходит то же самое, но команду DOS *type* нужно заменить командой Unix *cat*:

```
C:\...\PP2E\System\Streams>type input.txt
8
6

C:\...\PP2E\System\Streams>python teststreams.py < input.txt
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Здесь файл *input.txt* автоматизирует ввод данных, которые обычно поступают с клавиатуры в интерактивном режиме: вместо клавиатуры сценарий читает данные из этого файла. Аналогично можно переадресовать в файл стандартное устройство вывода с помощью синтаксиса оболочки > *filename*. На самом деле переадресацию ввода и вывода можно объединить в одной команде:

```
C:\...\PP2E\System\Streams>python teststreams.py < input.txt > output.txt

C:\...\PP2E\System\Streams>type output.txt
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

На этот раз как входные, так и выходные данные сценария Python отображаются на текстовые файлы, а не в сеанс интерактивной консоли.

Соединение программ с помощью каналов

В Windows и Unix-подобных системах можно направлять стандартный вывод одной программы в стандартный ввод другой, помещая между командами символ оболочки |. Обычно это называется операцией создания «канала» или «конвейера»: оболочка создает канал, соединяющий ввод и вывод двух команд. Отправим вывод сценария Python на вход программы командной строки «more», чтобы увидеть, как это действует:

```
C:\...\PP2E\System\Streams>python teststreams.py < input.txt | more
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Здесь стандартный ввод teststreams снова поступает из файла, но вывод (создаваемый утверждениями print) посылается другой программе, а не в файл или окно. Принимающей программой является more – стандартная программа командной строки для постраничного вывода, имеющаяся в Windows и Unix-подобных платформах. Поскольку Python привязывает сценарии к стандартной модели потоков, сценарии Python можно использовать с обоих концов: вывод одного сценария Python всегда можно отправить на ввод другого:

```
C:\...\PP2E\System\Streams>type writer.py
print "Help! Help! I'm being repressed!"
print 42

C:\...\PP2E\System\Streams>type reader.py
print 'Got this' "%s"' % raw_input()
import sys
data = sys.stdin.readline()[:-1]
print 'The meaning of life is', data, int(data) * 2

C:\...\PP2E\System\Streams>python writer.py | python reader.py
Got this" "Help! Help! I'm being repressed!"
The meaning of life is 42 84
```

На этот раз связаны между собой две программы Python. Сценарий reader получает входные данные из сценария writer; оба сценария просто читают и пишут в неведении о механизме потоков. На практике такое соединение программ в цепочку является простой формой связи между программами. Оно облегчает повторное использование утилит, написанных для связи через stdin и stdout, самыми неожиданными способами. Например, программу Python, которая сортирует текст, поступающий со stdin, можно применять к любому желаемому источнику данных, в том числе выводу других сценариев. Рассмотрим сценарии Python утилит командной строки из примеров 2.7 и 2.8, которые сортируют строки стандартного входного потока и складывают их.

Пример 2.7. PP2E\System\Streams\sorter.py

```
import sys
lines = sys.stdin.readlines()          # сортировать входные строки stdin,
lines.sort()                           # отправить результат на stdout
for line in lines: print line,         # для дальнейшей обработки
```

Пример 2.8. PP2E\System\Streams\adder.py

```
import sys, string
sum = 0
while 1:
    try:
        line = raw_input()               # или вызвать sys.stdin.readlines():
    except EOFError:                     # или цикл sys.stdin.readline()
        break
    else:
        sum = sum + string.atoi(line)  # int(line[:-1]) рассматривает 042 как восьмеричное число
print sum
```

Такие инструменты общего назначения можно применять в командной строке оболочки различными способами, чтобы сортировать и складывать произвольные файлы и выдачу программ:

```
C:\...\PP2E\System\Streams>type data.txt
123
000
```

```
999
042
C:\...\PP2E\System\Streams>python sorter.py < data.txt        сортировать файл
000
042
123
999
C:\...\PP2E\System\Streams>type data.txt | python adder.py    суммировать выдачу программы
1164
C:\...\PP2E\System\Streams>type writer2.py
for data in (123, 0, 999, 42):
    print '%03d' % data
C:\...\PP2E\System\Streams>python writer2.py | python sorter.py    сортировать вывод
000
042
123
999
C:\...\PP2E\System\Streams>python writer2.py | python sorter.py | python adder.py
1164
```

В последней команде с помощью стандартных потоков соединены три сценария Python: выход предыдущего сценария подается на вход последующего с помощью синтаксиса конвейера.

Если присмотреться, то можно заметить, что `sorter` читает сразу весь `stdin` с помощью метода `readlines`, но `adder` читает по одной строке. Если источником входных данных является другая программа, то на некоторых платформах соединенные каналом программы выполняются *параллельно*. В таких системах лучше производить построчное чтение, если пересылаемые потоки данных велики: читающей программе не приходится ждать, пока пишущая программа полностью завершит работу, чтобы заняться обработкой данных. Так как `raw_input` просто читает `stdin`, схему построчного ввода, используемую в `adder`, можно также закодировать с помощью `sys.stdin`:

```
C:\...\PP2E\System\Streams>type adder2.py
import sys, string
sum = 0
while 1:
    line = sys.stdin.readline()
    if not line: break
    sum = sum + string.atoi(line[:-1])
print sum
```

Однако перевод `sorter` на построчное чтение едва ли даст большой выигрыш в производительности, потому что метод списка `sort` требует, чтобы весь список был заполнен. Как будет показано в главе 17 «Структуры данных», вручную запрограммированные алгоритмы сортировки, скорее всего, будут работать значительно медленнее, чем метод сортировки списка Python.

Переадресованные потоки и взаимодействие с пользователем

В начале предыдущего раздела мы направили вывод *teststreams.py* на стандартную программу командной строки `more` с помощью такой команды:

```
C:\...\PP2E\System\Streams>python teststreams.py < input.txt | more
```

Но поскольку мы уже написали собственную утилиту постраничного вывода «more» на Python в начале этой главы, почему не сделать так, чтобы она тоже принимала

ввод из stdin? Например, изменим последние три строки файла *more.py*, приведенные ранее в этой главе, на следующие:

```
if __name__ == '__main__':                    # когда выполняется, а не импортируется
    if len(sys.argv) == 1:                    # выводить stdin, если нет аргументов
        more(sys.stdin.read())
    else:
        more(open(sys.argv[1]).read())
```

Тогда, похоже, мы сможем перенаправить стандартный вывод *teststreams.py* на стандартный ввод *more.py*:

```
C:\...\PP2E\System\Streams>python teststreams.py < input.txt | python ..\more.py
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

В целом такой прием действует для сценариев Python. Здесь *teststreams.py* снова принимает данные из файла. И, как и в предыдущем разделе, вывод одной программы Python отправляется по каналу на ввод другой – сценарий *more.py* в родительском («..») каталоге.

Чтение данных с клавиатуры. Однако в предыдущей команде *more.py* спрятана небольшая проблема. Действительно, цепочка сработала там по чистой случайности: если выдача первого сценария достаточно длинная и more придется спрашивать пользователя о продолжении, произойдет полный отказ сценария. Проблема в том, что улучшенный вариант *more.py* использует stdin с двумя различными целями. Он читает ответ пользователя со stdin, вызывая raw_input, но теперь еще и принимает со stdin основной входной текст. Когда поток stdin действительно переадресуется на входной файл или канал, его нельзя использовать для получения ответа от пользователя: он содержит только текст источника ввода. Кроме того, поскольку stdin переадресуется даже до начала программы, невозможно узнать, что он означал перед тем, как был переадресован в командной строке.

Если мы хотим принимать входные данные из stdin *и* использовать консоль для взаимодействия с пользователем, нужно сделать дополнительные изменения. В примере 2.9 показана модифицированная версия сценария more, в которой постранично выводится стандартный входной поток, если сценарий вызван без аргументов, но кроме того, используются низкоуровневые и зависящие от платформы средства, чтобы при необходимости общаться с пользователем за клавиатурой.

Пример 2.9. PP2E\System\moreplus.py

```
#############################################################################
# расщепить и интерактивно вывести на stdout строку, файл или поток текста;
# при запуске как сценарий выводить stdin или файл, имя которого передано
# в командной строке; если входом является stdin, нельзя использовать его для ответа
# пользователя - используйте платформо-зависимые средства для платформы или gui;
#############################################################################

import sys, string

def getreply():
    """
    считывает с консоли ответ пользователя
    даже если stdin переадресован на файл или поток
    """
    if sys.stdin.isatty():                    # если stdin является консолью,
```

```
        return raw_input('?')                  # прочитать строку ответа с stdin
    else:
        if sys.platform[:3] == 'win':          # если stdin переадресован,
            import msvcrt                      # нельзя использовать его для получения ответа
            msvcrt.putch('?')
            key = msvcrt.getche()              # используем средства консоли windows
            msvcrt.putch('\n')                 # getch() не выводит клавишу
            return key
        elif sys.platform[:5] == 'linux':      # используем устройство консоли linux
            print '?',                         # отрезать eoln в конце строки
            console = open('/dev/tty')
            line = console.readline()[:-1]
            return line
        else:
            print '[pause]'           # иначе просто приостановимся – сделайте лучше!
            import time                # смотрите также модули curses, tty
            time.sleep(5)              # или копируйте во врем. файл, перезапускайте
            return 'y'                 # или всплывающее окно gui, привязка клавиш tk

def more(text, numlines=10):
    """
    расщепить многострочный текст на stdout
    """
    lines = string.split(text, '\n')
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print line
        if lines and getreply() not in ['y', 'Y']: break

if __name__ == '__main__':                     # при выполнении, а не импорте
    if len(sys.argv) == 1:                     # если без аргументов командной строки
        more(sys.stdin.read())                 # выводить stdin, без raw_inputs
    else:
        more(open(sys.argv[1]).read())         # или выводить файл, переданный в аргументе
```

Большая часть нововведений этой версии кода находится в функции `getreply`. Метод файла `isatty` сообщает, соединен ли `stdin` с консолью; если да, просто считываем ответ со `stdin`, как и раньше. К сожалению, нет переносимого способа получить строку от пользователя консоли, минуя `stdin`, поэтому придется программировать ввод в зависимости от проверки `sys.platform`:

- В Windows встроенный модуль `msvcrt` предоставляет низкоуровневые вызовы для ввода и вывода на консоль (например, `msvcrt.getch()` считывает нажатие одной клавиши).
- В Linux файл системного устройства с именем */dev/tty* предоставляет доступ к данным, вводимым с клавиатуры (можно читать его как простой файл).
- На других платформах просто вызываем встроенную функцию `time.sleep`, чтобы сделать пятисекундную паузу между выводом (не идеально, но лучше, чем вообще не останавливаться; пусть будет так, пока не будет найдено лучшее непереносимое решение).

Конечно, такое изощренное программирование нужно только в сценариях, которые должны взаимодействовать с пользователями консоли *и* принимать данные на `stdin`. В приложении GUI, например, можно делать всплывающие диалоговые окна, привязывать события нажатия клавиш к обратным вызовам и т. д. (с GUI мы встретимся в главе 6 «Графические интерфейсы пользователя»).

Имея на вооружении функцию `getreply`, можно спокойно запускать утилиту `moreplus` различными способами. Как и прежде, можно импортировать и непосредственно вызывать функцию этого модуля, передавая ей ту строку, которую требуется вывести постранично:

```
>>> from moreplus import more
>>> more(open('System.txt').read())
This directory contains operating system interface examples.

Many of the examples in this unit appear elsewhere in the examples
distribution tree, because they are actually used to manage other
programs.  See the README.txt files in the subdirectories here for pointers.
```

И так же, как прежде, при запуске с *аргументом* командной строки этот сценарий интерактивно пролистывает текст указанного файла:

```
C:\...\PP2E\System>python moreplus.py System.txt
This directory contains operating system interface examples.

Many of the examples in this unit appear elsewhere in the examples
distribution tree, because they are actually used to manage other
programs. See the README.txt files in the subdirectories here for pointers.

C:\...\PP2E\System>python moreplus.py moreplus.py
##############################################################################
# расщепить и интерактивно вывести на stdout строку, файл или поток текста;
# при запуске как сценарий выводить stdin или файл, имя которого передано
# в командной строке; если входом является stdin, нельзя использовать его
# для получения ответа пользователя - используйте платформо-зависимые средства или gui;
##############################################################################
import sys, string

def getreply():
?n
```

Но теперь сценарий также правильно выводит постранично текст, переадресованный на `stdin` из файла или конвейером команд, даже если этот текст слишком велик для вывода целиком. В большинстве оболочек такой ввод посылается с помощью переадресации или операторов конвейера такого вида:

```
C:\...\PP2E\System>python moreplus.py < moreplus.py
##############################################################################
# расщепить и интерактивно вывести на stdout строку, файл или поток текста;
# при запуске как сценарий выводить stdin или файл, имя которого передано
# в командной строке; если входом является stdin, нельзя использовать его
# для получения ответа пользователя - используйте платформо-зависимые средства или gui;
##############################################################################
import sys, string

def getreply():
?n

C:\...\PP2E\System>type moreplus.py | python moreplus.py
##############################################################################
# расщепить и интерактивно вывести на stdout строку, файл или поток текста;
# при запуске как сценарий выводить stdin или файл, имя которого передано
# в командной строке; если входом является stdin, нельзя использовать его
# для получения ответа пользователя - используйте платформо-зависимые средства или gui;
##############################################################################

import sys, string
```

```
def getreply():
?n
```

Под Linux это действует так же, только вместо *type* используется команда *cat*. Наконец, если вывод одного сценария Python отправляется по каналу на ввод другого, все работает как надо, не нарушая взаимодействия с пользователем (и совсем не только потому, что нам повезло):

```
C:\.....\System\Streams>python teststreams.py < input.txt | python ..\moreplus.py
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Здесь стандартный *вывод* одного сценария Python подается на стандартный *ввод* другого сценария Python, находящегося в родительском каталоге: *moreplus.py* читает вывод *teststreams.py*.

Все переадресации в таких командных строках действуют только потому, что сценариям безразлично, чем в действительности являются стандартный ввод и вывод – пользователями, файлами или каналами, открытыми между программами. Например, при запуске в качестве сценария *moreplus.py* просто читает поток `sys.stdin`; оболочка командной строки (например, DOS в Windows, `csh` в Linux) прикрепляет такие потоки к источникам, определяемым командной строкой, перед запуском сценария. Сценарии используют для доступа к этим источникам заранее открытые объекты файлов `stdin` и `stdout` независимо от их истинной природы.

Для читателей, ведущих подсчет: мы запускали один этот сценарий постраничного вывода `more` четырьмя различными способами – путем импорта и вызова его функции, передавая имя файла в качестве аргумента, переадресуя `stdin` в файл и передавая выдачу команды по каналу в `stdin`. Благодаря поддержке импортируемых функций, аргументов командной строки и стандартных потоков код системных инструментов Python можно повторно использовать в разнообразных режимах.

Переадресация потоков объектам Python

Все приведенные выше способы переадресации стандартных потоков действуют для программ, написанных на любом языке, который перехватывает стандартные потоки, и зависят скорее от процессора командной строки оболочки, чем от самого Python. Такой синтаксис переадресации командной строки, как *<filename* и *|program*, обрабатывается оболочкой, а не Python. Более «питонистую» форму переадресации можно производить в самих сценариях, устанавливая `sys.stdin` и `sys.stdout` в объекты типа файлов.

Этот режим основывается на том, что любой объект, выглядящий по своим методам как файл, может работать в Python как стандартный поток. Важен не конкретный тип данных объекта, а его протокол. Это означает следующее:

- Любой объект, предоставляющий метод *read*, вроде того, который есть у файлов, может быть присвоен `sys.stdin`, в результате чего ввод будет осуществляться через методы чтения этого объекта.
- Любой объект, для которого определен метод *write* типа файлового, может быть присвоен `sys.stdout`; весь стандартный вывод будет отправляться методам этого объекта.

Так как `print` и `raw_input` просто вызывают методы `write` и `readline` тех объектов, на которые указывают `sys.stdout` и `sys.stdin`, можно создавать и перехватывать стандарт-

ные текстовые потоки с помощью объектов, реализованных как классы. Пример 2.10 показывает вспомогательный модуль, демонстрирующий эту идею.

Пример 2.10. PP2E\System\Streams\redirect.py

```
##############################################################################
# сходные с файлами объекты сохраняют весь текст стандартного вывода в строке
# и создают текст стандартного ввода из строки; redirect выполняет переданную функцию,
# для которой выходной и входной потоки установлены в эти похожие на файлы объекты классов;
##############################################################################
import sys, string                              # использовать встроенные модули

class Output:                                   # эмуляция выходного файла
    def __init__(self):
        self.text = ''                          # при создании – пустая строка
    def write(self, string):                    # добавить строку байтов
        self.text = self.text + string
    def writelines(self, lines):                # добавить все строки списка
        for line in lines: self.write(line)

class Input:                                    # эмуляция входного файла
    def __init__(self, input=''):               # аргумент по умолчанию
        self.text = input                       # сохранить строку после создания
    def read(self, *size):                      # необязательный аргумент
        if not size:                            # прочесть N байтов или все
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:size[0]], self.text[size[0]:]
        return res
    def readline(self):
        eoln = string.find(self.text, '\n')     # найти смещение следующего eoln
        if eoln == -1:                          # обрезать до eoln
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:eoln+1], self.text[eoln+1:]
        return res

def redirect(function, args, input):            # перенаправить stdin/out
    savestreams = sys.stdin, sys.stdout         # выполнить объект функции
    sys.stdin   = Input(input)                  # возвратить текст stdout
    sys.stdout  = Output()
    try:
        apply(function, args)
    except:
        sys.stderr.write('error in function! ')
        sys.stderr.write("%s, %s\n" % (sys.exc_type, sys.exc_value))
    result = sys.stdout.text
    sys.stdin, sys.stdout = savestreams
    return result
```

В этом модуле определены два класса, маскирующиеся под настоящие файлы:

- `Output` предоставляет протокол метода записи, предполагаемый у выходных файлов, но сохраняет всю записываемую выдачу в строке, хранящейся в памяти.
- `Input` предоставляет протокол, предполагаемый у входных файлов, но обеспечивает ввод по требованию из хранящейся в памяти строки, переданной при создании объекта.

Функция `redirect` в конце этого файла объединяет эти два объекта, чтобы выполнять одну функцию для ввода и вывода полностью переадресованных объектам классов

Python. Переданной функции, выполняемой таким образом, безразлично, что ее утверждения `print`, вызовы `raw_input` и вызовы методов для `stdin` и `stdout` имеют дело с классом, а не с действительным файлом, каналом или пользователем.

Для демонстрации этого сделаем импорт и запустим функцию `interact`, лежащую в основе сценария `teststreams`, запускаемого из оболочки (для использования вспомогательной функции переадресации нужно действовать на языке функций, а не файлов). При непосредственном запуске функция производит чтение с клавиатуры и запись на экран, как если бы она выполнялась как программа без переадресации:

```
C:\...\PP2E\System\Streams>python
>>> from teststreams import interact
>>> interact()
Hello stream world
Enter a number>2
2 squared is 4
Enter a number>3
3 squared is 9
Enter a number
>>>
```

Теперь запустим эту функцию под управлением функции переадресации в *redirect.py* и передадим некоторый готовый входной текст. В таком режиме на вход функции `interact` поступает переданная строка (`'4\n5\n6\n'` — три строчки с явными символами конца строки), а результатом выполнения функции будет строка, содержащая весь текст, выводимый в стандартный выходной поток:

```
>>> from redirect import redirect
>>> output = redirect(interact, (), '4\n5\n6\n')
>>> output
'Hello stream world\012Enter a number>4 squared is 16\012Enter a number>
5 squared is 25\012Enter a number>6 squared is 36\012Enter a number>Bye\012'
```

Результатом является одна длинная строка, в которой конкатенирован весь текст, записываемый в стандартный вывод. Улучшить внешний вид можно, расщепив ее с использованием стандартного модуля `string`:

```
>>> from string import split
>>> for line in split(output, '\n'): print line
...
Hello stream world
Enter a number>4 squared is 16
Enter a number>5 squared is 25
Enter a number>6 squared is 36
Enter a number>Bye
```

Еще лучше повторно использовать модуль *more.py*, который мы видели ранее в этой главе: вводить и помнить меньше, а качество работы проверено:

```
>>> from PP2E.System.more import more
>>> more(output)
Hello stream world
Enter a number>4 squared is 16
Enter a number>5 squared is 25
Enter a number>6 squared is 36
Enter a number>Bye
```

Конечно, это искусственный пример, но показанные приемы могут иметь широкое применение. Например, в программу, написанную для взаимодействия с пользовате-

лем в командной строке, легко добавить графический интерфейс. Нужно просто перехватить стандартный ввод с помощью объекта типа показанного класса Output и сбросить текстовую строку в окно. Аналогично стандартный ввод можно переустановить в объект, который получает текст из графического интерфейса (например, всплывающего диалогового окна). Поскольку классы могут заменять реальные файлы, их можно использовать в любых инструментах, работающих с файлами. Смотрите модуль переадресации потоков с GUI под именем guiStreams в главе 9.

Другие варианты переадресации

Ранее в этой главе мы изучали также встроенную функцию os.popen, предоставляющую возможность переадресации потоков других команд из программы Python. Как мы видели, эта функция выполняет командную строку оболочки (например, строку, которая обычно вводится с клавиатуры в ответ на приглашение DOS или csh), но возвращает объект Python типа файла, соединенный с входным или выходным потоком команды. Благодаря этому os.popen можно рассматривать как еще один способ переадресации потоков порождаемых программ, родственный только что рассмотренным приемам: ее действие во многом похоже на синтаксис команды конвейеризации оболочки («|», в действительности ее имя означает «pipe open» – «канал открыт»), но она выполняется внутри сценария и предоставляет схожий с файловым интерфейс к потоку данных в канале. По духу она близка функции redirect, но основана на выполнении программ, а не функций, а ее потоки обрабатываются в порождающем сценарии как файлы (не привязанные к объектам классов).

Передавая в функцию флаг нужного режима, мы переадресуем в вызывающих сценариях входной или выходной поток в файлы:

```
C:\...\PP2E\System\Streams>type hello-out.py
print 'Hello shell world'
C:\...\PP2E\System\Streams>type hello-in.py
input = raw_input()
open('hello-in.txt', 'w').write('Hello ' + input + '\n')
C:\...\PP2E\System\Streams>python
>>> import os
>>> pipe = os.popen('python hello-out.py')         # 'r' флаг по умолчанию - читать из stdout
>>> pipe.read()
'Hello shell world\012'
>>> pipe = os.popen('python hello-in.py', 'w')
>>> pipe.write('Gumby\n')                          # 'w'-писать в stdin программы
>>> pipe.close()                                   # \n в конце необязателен
>>> open('hello-in.txt').read()
'Hello Gumby\012'
```

Вызов popen также достаточно сообразителен, чтобы выполнять командную строку как независимый процесс в Unix и Linux. В библиотеке Python есть дополнительные средства типа popen, позволяющие сценариям соединяться более чем с одним из потоков команды. Например, в модуле popen2 есть функции для перехвата одновременно входного и выходного потоков команды (popen2.popen2), а также для соединения со стандартным устройством ошибок (popen2.popen3):

```
import popen2
childStdout, childStdin = popen2.popen2('python hello-in-out.py')
childStdin.write(input)
output = childStdout.read()

childStdout, childStdin, childStderr = popen2.popen3('python hello-in-out.py')
```

Эти два вызова действуют во многом подобно `os.popen`, но подключают дополнительные потоки. Когда я впервые писал об этом, данные вызовы работали только на Unix-подобных платформах, но не в Windows, поскольку в Python 1.5.2 они использовали вызов `fork`. Теперь, в версии Python 2.0, они работают и в Windows.

По этому поводу надо заметить, что на Unix-подобных платформах с помощью комбинации вызовов `os.fork`, `os.pipe`, `os.dup` и некоторых вариантов `os.exec` можно запустить новую независимую программу, потоки которой соединены с потоками родительской программы (так работает `popen2`). Стало быть, это еще один способ переадресации потоков и низкоуровневый эквивалент таких средств, как `os.popen`. Подробнее об этих вызовах читайте в главе 3, особенно в разделе по каналам.

В Python 2.0 вызовы `popen2` и `popen3` теперь доступны через модуль `os`. (Например, `os.popen2` – то же самое, что `popen2.popen2`, за исключением того, что в результате вызова `stdin` и `stdout` поменялись местами.) Кроме того, в версии 2.0 расширено утверждение `print`, чтобы можно было явно указать файл, в который направляется выдача. Утверждение вида `print >>file stuff` выводит `stuff` в `file` вместо `stdout`. Получаемый результат аналогичен простому присвоению `sys.stdout` объекту.

Перехват потока stderr

Мы сосредоточились на переадресации `stdin` и `stdout`, но `stderr` можно аналогичным образом переадресовывать в файлы, каналы и объекты. В сценарии Python это делается просто. Например, присвоение `sys.stderr` другому экземпляру класса, вроде `Output` из предыдущего примера, позволяет сценарию перехватывать также текст, записываемый в стандартное устройство ошибок. Упомянутый выше вызов `popen3` также позволяет перехватывать в сценарии `stderr`.

Переадресация стандартного потока ошибок из командной строки оболочки несколько более сложна и хуже переносится. В большинстве Unix-подобных систем обычно можно перехватить вывод в `stderr` с помощью синтаксиса переадресации оболочки вида `command 2>&1`. Однако на платформах Windows 9x он не действует и может различаться даже в зависимости от оболочки Unix; смотрите подробности в страницах руководства по вашей оболочке.

Средства для работы с файлами

В основном утилиты оболочки предназначаются для работы с внешними файлами. Например, тестирующая система может читать входные данные из одного файла, сохранять результаты программы в другом файле и проверять полученные результаты, загружая третий файл. Даже программы интерфейса пользователя и Интернет-ориентированные программы могут загружать двоичные изображения и аудиоклипы из локальных файлов. Это базовая концепция программирования.

В Python главным инструментом, используемым сценариями для доступа к файлам компьютера, служит встроенная функция `open`. Поскольку эта функция является неотъемлемой частью языка Python, вы, возможно, уже знакомы с основами ее работы. Технически `open` предоставляет прямой доступ к вызовам файловой системы `stdio` в С-библиотеке системы: она возвращает новый объект файла, соединенный с внешним файлом и обладающий методами, более или менее непосредственно связанными с системными файловыми вызовами на вашей машине. Функция `open` предоставляет так-

же переносимый интерфейс к используемой файловой системе – она одинаково работает на любой платформе, где выполняется Python.

Другие относящиеся к файлам интерфейсы в Python позволяют делать такие вещи, как работа с файлами на низком уровне с использованием файловых дескрипторов (модуль `os`), сохранять в файлах объекты по ключу (модули `anydbm` и `shelve`) и обращаться к базам данных SQL. Это, по большей части, крупные темы, о которых говорится в главе 16 «Базы данных и постоянное хранение». В данном разделе мы кратко рассмотрим встроенный объект файла и несколько более сложных тем, относящихся к файлам. Как обычно, более подробное описание и методы, для рассказа о которых здесь недостаточно места, следует искать в описании файловых объектов в руководстве по библиотеке.

Встроенные объекты файлов

Для большинства задач обработки файлов в сценариях достаточно знать функцию `open`. У объекта файла, возвращаемого `open`, есть методы для чтения данных (`read`, `readline`, `readlines`), записи данных (`write`, `writelines`), освобождения системных ресурсов (`close`), перемещения по файлу (`seek`), принудительной очистки буферов (`flush`), получения соответствующего дескриптора файла (`fileno`) и другие. Однако ввиду простоты использования встроенного объекта файла сразу перейдем к некоторым интерактивным примерам.

Вывод в файлы

Чтобы создать новый файл, вызовите `open` с двумя аргументами: внешним именем создаваемого файла и строкой режима «w» (сокращенно от «write»). Чтобы сохранить данные в файле, вызовите метод `write` объекта файла со строкой, содержащей данные, которые нужно сохранить, а затем метод `close`, чтобы закрыть файл, если вы хотите снова открыть его в той же программе или сеансе:

```
C:\temp>python
>>> file = open('data.txt', 'w')          # открыть объект выходного файла: создание
>>> file.write('Hello file world!\n')     # пишет строку как есть
>>> file.write('Bye   file world.\n')
>>> file.close()                          # закрыть и выйти
```

Вот и все – вы только что создали на своем компьютере, неважно каком, совершенно новый файл:

```
C:\temp>dir data.txt /B
data.txt

C:\temp>type data.txt
Hello file world!
Bye   file world.
```

В новом файле нет ничего необычного. Здесь для показа имени файла и отображения его содержимого использованы команды DOS *dir* и *type*, но в GUI менеджера файлов он тоже будет виден.

Открытие файлов. В вызове функции `open`, показанном в предыдущем примере, первый аргумент может задавать в составе строки с именем файла необязательный полный путь к каталогу. Если просто передать имя файла без указания пути, файл окажется в текущем рабочем каталоге Python. То есть он появится в том месте, в котором выполняется код, – в данном случае голое имя файла *data.txt* влечет использование каталога *C:\temp* на моей машине, поэтому в действительности создается файл

C:\temp\data.txt. Освежить эту тему в памяти можно с помощью раздела «Текущий рабочий каталог» выше в этой главе.

Обратите также внимание, что при открытии в режиме «w» Python либо создает новый файл, если он еще не существует, либо стирает текущее содержимое файла, если он уже есть на машине (поэтому следует проявлять осторожность).

Запись. Обратите внимание, что в строки, записываемые в файл, был явно добавлен символ конца строки `\n`; в отличие от утверждения `print`, методы файлов `write` записывают в точности то, что им передано, без дополнительного форматирования. Строка, переданная `write`, появляется во внешнем файле «байт в байт».

Выходные файлы предоставляют также метод `writelines`, который просто записывает все строки списка по одной без дополнительного форматирования. Например, вот вызов `writelines`, эквивалентный двум вызовам `write`, показанным ранее:

```
file.writelines(['Hello file world!\n', 'Bye   file world.\n'])
```

Этот вызов используется не так часто (и может эмулироваться простым циклом `for`), но может быть удобен в сценариях, которые сначала сохраняют выходные данные в списке, а потом записывают его.

Закрытие. Использованный выше метод файла `close` завершает формирование содержимого файла и освобождает системные ресурсы. Например, закрытие файла влечет сброс на диск буферизованных выходных данных. Обычно файлы автоматически закрываются при уборке интерпретатором объекта файла как мусора (то есть когда на него не оказывается больше ссылок) и при завершении сеанса или программы Python. Благодаря этому вызовы `close` часто оказываются необязательными. В действительности часто можно увидеть такой код Python для обработки файла:

```
open('somefile.txt').write("G'day Bruce\n")
```

Так как это выражение создает временный объект файла, сразу осуществляет в него запись и не сохраняет ссылки на него, объект файла немедленно закрывается и уничтожается без всякого явного вызова метода `close`.

Обратите внимание, что существует возможность изменения этой функции *автоматического закрытия при удалении* объектов файлов в будущих версиях Python. Более того, JPython – основанная на Java реализация Python, обсуждаемая ниже, не уничтожает сразу файловые объекты, как это делает стандартная система Python (в ней используется сборщик мусора Java). Если в вашем сценарии создается много файлов, а платформа ограничивает число открытых файлов в каждой программе, полезной привычкой будет явный вызов `close`.

Чтение из файлов

Чтение данных из внешних файлов осуществляется столь же просто, как запись, но методы, позволяющие загружать данные в разнообразных режимах, многочисленнее. Входные текстовые файлы открываются с флагом режима «r» (от «read») либо вообще без флага режима (значением режима по умолчанию является «r»). После открытия текстового файла его строки можно читать с помощью метода `readlines`:

```
>>> file = open('data.txt', 'r')          # открыть объект входного файла
>>> for line in file.readlines():         # читать в список строк
...     print line,                       # в конце строк стоит '\n'
...
```

```
Hello file world!
Bye   file world.
```

Метод readlines загружает целиком содержимое файла в память и предоставляет его сценарию в виде списка строк, который можно обойти в цикле. В действительности существует много способов чтения входного файла:

- file.read() возвращает строку, содержащую все байты, хранящиеся в файле.
- file.read(N) возвращает строку, содержащую очередные N байтов файла.
- file.readline() читает до очередного \n и возвращает строку.
- file.readlines() читает файл целиком и возвращает список строк.

Выполним эти методы для чтения файлов, строк и байтов:

```
>>> file.seek(0)                                # вернуться в начало файла
>>> file.read()                                 # прочесть весь файл в строку
'Hello file world!\012Bye   file world.\012'

>>> file.seek(0)
>>> file.readlines()
['Hello file world!\012', 'Bye   file world.\012']

>>> file.seek(0)
>>> file.readline()
'Hello file world!\012'
>>> file.readline()
'Bye   file world.\012'

>>> file.seek(0)
>>> file.read(1), file.read(8)
('H', 'ello fil')
```

Все эти методы ввода позволяют определить, сколько данных должно быть получено. Вот несколько практических правил относительно выбора метода:

- read() и readlines() загружают в память сразу *весь файл*. Это удобно для получения содержимого файла с помощью возможно более короткого кода. Кроме того, эти методы действуют очень быстро, но для больших файлов их применение накладно: загрузка гигабайтных файлов в память обычно не оправданна.
- С другой стороны, поскольку вызовы readline() и read(N) получают лишь *часть файла* (очередную строку или блок из N байтов), они надежнее для потенциально больших файлов, но несколько менее удобны и обычно работают гораздо медленнее. Если скорость важна и ваши файлы не слишком велики, лучшим выбором может оказаться read или readlines.

Между прочим, часто встречающийся здесь вызов seek(0) означает «вернуться в начало файла». Все операции чтения и записи в файлах происходят в текущей позиции; обычно при открытии файлы начинаются со смещения 0, которое перемещается вперед по мере передачи данных. Вызов seek просто позволяет переместиться в новое положение для очередной операции передачи данных. В Python метод seek принимает также необязательный второй аргумент, имеющий одно из трех значений: 0 – для абсолютной позиции в файле (по умолчанию), 1 – для перемещения относительно текущей позиции и 2 – для перемещения относительно конца файла. Когда в seek передается только аргумент смещения 0, как выше, это примерно соответствует операции *обратной перемотки (rewind)* файла.

Другие режимы файлового объекта

Помимо «w» и «r» большинством платформ поддерживается строка режима открытия «a», означающая «append» (дополнение). В этом режиме вывода методы `write` добавляют данные в конец файла, и вызов `open` не стирает текущее содержимое файла:

```
>>> file = open('data.txt', 'a')          # открытие в режиме дописывания:
                                          # содержимое не стирается
>>> file.write('The Life of Brian')       # добавляется в конец существующих данных
>>> file.close()
>>>
>>> open('data.txt').read()               # открыть файл и читать целиком
'Hello file world!\012Bye   file world.\012The Life of Brian'
```

Большинство файлов открывается с использованием такого рода вызовов, но `open` фактически допускает задание до трех аргументов для более специфических потребностей обработки – имя файла, режим открытия и размер буфера. Все они, кроме первого, являются необязательными: если они опущены, принимается режим открытия по умолчанию «r» (ввод), а политика в отношении размера буфера на большинстве платформ разрешает буферизацию. Вот некоторые вещи, которые следует знать обо всех трех аргументах `open`:

Имя файла

Как уже говорилось, имена файлов могут включать путь к каталогу, чтобы ссылаться на файлы, находящиеся на компьютере в произвольном месте; если путь не включен, имена считаются заданными относительно текущего рабочего каталога (описанного выше). В целом, любой формат имени файла, который можно ввести в системной оболочке, годится в вызове `open`. Например, аргумент имени файла `r'..\temp\spam.txt'` в Windows означает файл *spam.txt* в подкаталоге *temp* родительского каталога текущего рабочего каталога – на один шаг вверх и затем вниз в каталог *temp*.

Режим открытия

Для функции `open` можно задавать и другие режимы, некоторые из которых не показаны в данной книге (например, `r+`, `w+` и `a+` при открытии для обновления и «b» для обозначения двоичного режима). В частности, режим `r+` означает, что файл доступен как для чтения, так и для записи, а `wb` записывает данные в двоичном режиме (подробнее об этом в следующем разделе). В целом, все, что можно использовать в качестве строки режима в вызове `fopen` языка C на вашей платформе, будет работать в функции `open` Python, поскольку она в действительности просто внутренне вызывает `fopen`. (Если вы не знаете C, не особо страдайте по этому поводу.) Обратите внимание, что в программах Python содержимым файлов всегда являются строки, независимо от режима открытия: строку возвращают методы чтения, и строку мы передаем методам записи.

Размер буфера

Вызов `open` также принимает необязательный третий аргумент с *размером буфера*, позволяющий управлять буферизацией `stdio` для файла – способом размещения данных в очереди, позволяющим повысить производительность. Передача 0 означает отсутствие буферизации (данные передаются немедленно), 1 означает построчную буферизацию, любое другое положительное число означает использование буфера примерно указанного размера, а отрицательное значение означает использование величины, установленной в системе по умолчанию (которое используется и при отсутствии третьего аргумента и в целом означает, что буферизация включена). Аргумент размера буфера работает на большинстве платформ, но в

настоящее время игнорируется на платформах, не поддерживающих системный вызов sevbuf.

Файлы двоичных данных

Во всех предыдущих примерах обрабатываются простые текстовые файлы. На большинстве платформ сценарии Python могут также открывать и обрабатывать файлы, содержащие *двоичные* данные – изображения JPEG, аудиоклипы и все остальное, что можно хранить в файлах. Главное отличие кода состоит в аргументе *режима*, передаваемом встроенной функции open:

```
>>> file = open('data.txt', 'wb')      # открыть двоичный файл для вывода
>>> file = open('data.txt', 'rb')      # открыть двоичный файл для чтения
```

После открытия двоичных файлов таким способом можно читать и записывать их содержимое с помощью тех же самых только что показанных методов: read, write и т. д. (Однако readline и readlines не имеют здесь смысла: двоичные данные не являются строковыми.)

Во всех случаях данные, перемещаемые между файлами и программами, представляются в сценариях виде *строк* Python, даже если они являются двоичными. Это действует, поскольку строковые объекты Python могут содержать байты символов с любыми значениями (даже если при выводе они выглядят необычно). Интересно, что в строке Python могут содержаться даже байты со значениями 0; в обозначениях escape-кодов они выглядят как \0 и не завершают строк в Python, как это происходит в C. Например:

```
>>> data = "a\0b\0c"
>>> data
'a\000b\000c'
>>> len(data)
5
```

Не полагаясь на терминатор строки, Python хранит длину строки явным образом. Здесь data ссылается на строку длины 5, в которой оказалось два байта со значением ноль; они выводятся в формате восьмеричного кода как \000. Поскольку зарезервированных кодов символов нет, допускается читать двоичные данные с нулевыми байтами (и другими значениями) в строку Python.

Трансляция конца строки в Windows

Строго говоря, на некоторых платформах для обработки двоичных файлов может не требоваться «b» в конце аргумента режима открытия файла; «b» просто игнорируется, поэтому режимы «r» и «w» работают с тем же успехом. В действительности «b» в строках флага режима обычно требуется только для двоичных файлов в Windows. Чтобы разобраться почему, необходимо знать, как завершаются строки в текстовых файлах.

По историческим причинам конец строки текста в файле представляется на разных платформах различными символами: в Unix и Linux это одиночный символ \n, но в Windows это последовательность из двух символов \r\n.[1] В результате файлы, перемещаемые между Linux и Windows, могут после передачи странно выглядеть в тексто-

[1] Фактически дело еще хуже: на Мас строки текстовых файлов завершаются одиночным \r (не \n или \r\n). Тот, кто сказал, что патентованное программное обеспечение – это хорошо для потребителя, видимо, не имел в виду пользователей нескольких платформ и точно не учитывал программистов.

вом редакторе: они могут сохранить окончание строки, принятое на исходной платформе. Например, большинство редакторов Windows обрабатывает текст в формате Unix, но Notepad (блокнот) составляет заметное исключение: текстовые файлы, скопированные из Unix или Linux, обычно выглядят в Notepad как одна большая строка со странными символами внутри (\n).

Сценариям Python это обычно безразлично, поскольку перенос под Windows (в действительности действующий в Windows компилятор C) автоматически отображает последовательность DOS \r\n в одиночный \n. При выполнении сценариев в Windows это действует так:

- Для файлов, открытых в текстовом режиме, \r\n транслируется при вводе в \n.
- Для файлов, открытых в текстовом режиме, \n транслируется при выводе в \r\n.
- Для файлов, открытых в двоичном режиме, при вводе или выводе не производится трансляция.
- На Unix-подобных платформах трансляция не происходит в любом режиме открытия.

Следует запомнить два важных следствия из этих правил. Во-первых, почти всегда во всех сценариях Python символ конца строки представляется одиночным \n, независимо от способа его сохранения во внешних файлах на соответствующей платформе. Путем соответствующей трансляции \n при вводе и выводе перенос под Windows скрывает присущие платформам различия.

Второе следствие этой трансляции более тонкое: если вы собираетесь обрабатывать *файлы двоичных данных* под Windows, то обычно должны следить за открытием этих файлов в двоичном («rb», «wb»), а не текстовом («r», «w») режиме. В противном случае указанные преобразования вполне могут повредить данные как при вводе, так и при выводе. Вполне возможно, что среди двоичных данных окажутся такие байты, как символы конца строки DOS, то есть \r и \n. Если обрабатывать подобные двоичные файлы в *текстовом* режиме под Windows, байты \r могут быть ошибочно отброшены при вводе, а байты \n ошибочно превращены в \r\n при записи. В итоге двоичные данные окажутся искаженными, что, вероятно, вам совсем не нужно! Например под Windows:

```
>>> len('a\0b\rc\r\nd')                          # 4 байта escape-кодов
8
>>> open('temp.bin', 'wb').write('a\0b\rc\r\nd')  # запись двоичных данных в файл

>>> open('temp.bin', 'rb').read()                 # верно при двоичном чтении
'a\000b\015c\015\012d'

>>> open('temp.bin', 'r').read()                  # теряет \r в текстовом режиме!
'a\000b\015c\012d'

>>> open('temp.bin', 'w').write('a\0b\rc\r\nd')   # добавляет \r в текстовом режиме!
>>> open('temp.bin', 'rb').read()
'a\000b\015c\015\015\012d'
```

Эта проблема возникает только при работе под Windows, но использование с двоичными файлами режимов двоичного открытия «rb» и «wb» не повредит и на других платформах, а ваши сценарии станут более переносимыми (как знать, может быть, утилита для Unix найдет применение на вашем PC).

Есть и другие случаи, когда может потребоваться использование режимов открытия двоичных файлов. Например, в главе 5 «Более крупные системные примеры, часть 2» мы встретимся со сценарием под именем fixeoln_one, который транслирует символы окончания строки в текстовых файлах между DOS и Unix. Такой сценарий тоже дол-

жен открывать *текстовые* файлы в *двоичном* режиме, чтобы видеть, какие символы конца строки действительно присутствуют в файле: в текстовом режиме они уже были бы оттранслированы в \n ко времени поступления в сценарий.

Средства работы с файлами в модуле os

Модуль os содержит дополнительный набор функций обработки файлов, отличных от средств работы со встроенными *объектами* файлов, продемонстрированными в предыдущих примерах. Вот, например, весьма неполный список относящихся к файлам вызовов os:

`os.open(path, flags, mode)`

Открывает файл, возвращает его описатель

`os.read(descriptor, N)`

Читает не более *N* байтов, возвращает строку

`os.write(descriptor, string)`

Пишет байты из *string* в файл

`os.lseek(descriptor, position)`

Перемещается в файле на позицию *position*

Технически вызовы os обрабатывают файлы по их дескрипторам – целочисленным кодам или «описателям» (handles), идентифицирующим файлы в операционной системе. Поскольку средства работы с файлами с использованием дескрипторов в os имеют более низкий уровень и более сложны, чем встроенные объекты файлов, создаваемые с помощью встроенной функции open, обычно следует использовать последние, за исключением очень специфических потребностей обработки файлов.[1]

Однако чтобы дать вам общее представление об этом наборе инструментов, проведем несколько интерактивных экспериментов. Хотя встроенные объекты файлов и файловые дескрипторы модуля os обрабатываются различными наборами инструментов, они в действительности связаны между собой: файловая система stdio, используемая файловыми объектами, просто накладывает дополнительную логику поверх дескрипторов файлов.

В действительности метод объекта файла fileno возвращает целочисленный дескриптор, ассоциируемый со встроенным объектом файла. Например, файловые объекты стандартных потоков имеют дескрипторы 0, 1 и 2; вызов функции os.write для отправки данных на stdout по дескриптору имеет такой же результат, как вызов метода sys.stdout.write:

```
>>> import sys
>>> for stream in (sys.stdin, sys.stdout, sys.stderr):
...     print stream.fileno(),
...
0 1 2
>>> sys.stdout.write('Hello stdio world\n')        # запись через метод файла
Hello stdio world
>>> import os
>>> os.write(1, 'Hello descriptor world\n')        # запись через модуль os
Hello descriptor world
23
```

[1] Например, для обработки каналов (*pipes*), описываемых в главе 3. В Python вызов pipe возвращает два файловых дескриптора, которые можно обрабатывать средствами модуля os или заключить в файловый объект с помощью os.fdopen.

Поскольку объекты файлов, открываемые явно, ведут себя одинаково, можно обрабатывать данный реальный внешний файл соответствующего компьютера через встроенную функцию open, средства модуля os или то и другое вместе:

```
>>> file = open(r'C:\temp\spam.txt', 'w')          # создать внешний файл
>>> file.write('Hello stdio file\n')               # запись через метод файла
>>>
>>> fd = file.fileno()
>>> print fd
3
>>> os.write(fd, 'Hello descriptor file\n')        # запись через модуль os
22
>>> file.close()
>>>
C:\WINDOWS>type c:\temp\spam.txt                   # видны та и другая записи
Hello descriptor file
Hello stdio file
```

Флаги режима открытия

Зачем же нужны дополнительные файловые средства в os? Если вкратце, то они позволяют дополнительно управлять обработкой файлов на низком уровне. Встроенную функцию open просто использовать, но она ограничена файловой системой stdio, для которой является оболочкой: буферизация, режимы открытия и так далее определяются установками stdio по умолчанию.[1] Модуль os позволяет сценариям быть более точными; например, следующий код открывает файл через дескриптор в двоичном режиме для чтения-записи, осуществляя побитовое «ИЛИ» над двумя флагами режима, экспортируемыми os:

```
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> os.read(fdfile, 20)
'Hello descriptor fil'
>>> os.lseek(fdfile, 0, 0)                         # вернуться в начало файла
0
>>> os.read(fdfile, 100)                           # двоичный режим сохраняет '\r\n'
'Hello descriptor file\015\012Hello stdio file\015\012'

>>> os.lseek(fdfile, 0, 0)
0
>>> os.write(fdfile, 'HELLO')                      # перезаписать первые 5 байтов
5
```

В некоторых системах такие флаги открытия позволяют задавать при открытии файла более сложные режимы типа *исключительного доступа* (O_EXCL) и *неблокирующего* режима (O_NONBLOCK). Некоторые из этих флагов не переносимы между платформами (еще одно основание использовать преимущественно встроенные объекты файлов); смотрите руководство по библиотеке или выполните на своей машине вызов dir(os), чтобы получить полный список других имеющихся флагов открытия.

Ранее было показано, как перейти от объекта файла к дескриптору с помощью метода объекта файла fileno. Можно пойти и обратным путем: вызов os.fdopen помещает дескриптор файла в объект файла. Поскольку преобразования действуют в обоих на-

[1] Чтобы быть честными по отношению к объекту файла, следует упомянуть, что функция open может принимать флаг режима «rb+», что эквивалентно используемым здесь объединенным флагам режима, и также может открывать файл без буферизации, если значение аргумента размера буфера равно 0 . Так что по возможности лучше используйте open, а не os.open.

правлениях, обычно можно использовать любой набор средств – объект файла или модуль os:

```
>>> objfile = os.fdopen(fdfile)
>>> objfile.seek(0)
>>> objfile.read()
'HELLO descriptor file\015\012Hello stdio file\015\012'
```

Другие файловые средства os

В модуль os входит также ряд файловых инструментов, которые принимают строку пути к файлу и выполняют ряд связанных с файлами задач, таких как переименование (os.rename), удаление (os.remove) и изменение владельца файла и прав доступа к нему (os.chown, os.chmod). Рассмотрим несколько примеров использования этих средств:

```
>>> os.chmod('spam.txt', 0777)          # разрешен любой вид доступа
```

Этот вызов os.chmod установки прав доступа к файлу передает строку из девяти битов, составленную из трех наборов по три бита каждый. Эти три набора представляют, слева направо, пользователя-владельца файла, группу файла и всех остальных. Образующие каждый набор три бита отражают доступ по чтению, записи и выполнению. Если в этой строке бит равен «1», это означает разрешение соответствующей операции. Например, восьмеричное 0777 является строкой из девяти битов «1» в двоичном представлении, что разрешает все три вида доступа для всех трех групп пользователей; восьмеричное 0600 означает возможность только чтения и записи для пользователя, который им владеет, восьмеричное 0600 в двоичной записи дает 110 000 000.

Эта схема ведет свое происхождение от прав доступа в Unix, но работает также в Windows. Если она вас озадачила, посмотрите страницу руководства Unix для *chmod* или практический пример fixreadonly в главе 5 «Более крупные системные примеры, часть 2» (открытие на запись файлов только для чтения, копируемых с CD-ROM).

```
>>> os.rename(r'C:\temp\spam.txt', r'C:\temp\eggs.txt')      # (из, в)
>>>
>>> os.remove(r'C:\temp\spam.txt')                          # удалить файл
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OSError: [Errno 2] No such file or directory: 'C:\\temp\\spam.txt'
>>>
>>> os.remove(r'C:\temp\eggs.txt')
```

Использованный здесь вызов os.rename изменяет имя файла; вызов os.remove удаляет файл с машины и служит синонимом для os.unlink; последний отражает имя, которое имеет эта команда в Unix, но оно непонятно пользователям других платформ. Модуль os также экспортирует системный вызов stat:

```
>>> import os
>>> info = os.stat(r'C:\temp\spam.txt')
>>> info
(33206, 0, 2, 1, 0, 0, 41, 968133600, 968176258, 968176193)

>>> import stat
>>> info[stat.ST_MODE], info[stat.ST_SIZE]
(33206, 41)

>>> mode = info[stat.ST_MODE]
>>> stat.S_ISDIR(mode), stat.S_ISREG(mode)
(0, 1)
```

Вызов os.stat возвращает набор величин, представляющих низкоуровневую информацию о файле с заданным именем, а модуль stat экспортирует константы и функции для запроса этой информации переносимым образом. Например, значение, получаемое из результата os.stat по индексу stat.ST_SIZE, возвращает размер файла, а вызвав stat.S_ISDIR с элементом режима из результата os.stat можно проверить, является ли файл каталогом. Однако, как было показано раньше, обе эти операции доступны и через модуль os.path, поэтому в использовании os.stat редко возникает необходимость, кроме как для файловых запросов низкого уровня:

```
>>> path = r'C:\temp\spam.txt'
>>> os.path.isdir(path), os.path.isfile(path), os.path.getsize(path)
(0, 1, 41)
```

Сканеры файлов

В отличие от некоторых инструментальных языков оболочки, в Python нет явной процедуры циклического сканирования файла, но можно легко написать такую общую процедуру и использовать ее многократно. Модуль в примере 2.11 определяет общую процедуру сканирования файлов, которая применяет переданную в нее функцию Python к каждой строчке внешнего файла.

Пример 2.11. PP2E\System\Filetools\scanfile.py

```
def scanner(name, function):
    file = open(name, 'r')              # создать объект файла
    while 1:
        line = file.readline()          # вызов методов файла
        if not line: break              # до конца файла
        function(line)                  # вызов объекта функции
    file.close()
```

Функции scanner безразлично, какая функция обработки строк в нее передана, чем, в основном и определяется ее общность: она готова применить *любую* функцию одного аргумента, уже существующую или могущую возникнуть в будущем, ко всем строчкам текстового файла. Если код этого модуля поместить в каталог, входящий в PYTHONPATH, им можно воспользоваться всякий раз, когда требуется пройти по файлу строка за строкой. Пример 2.12 показывает сценарий клиента, выполняющего простую трансляцию строк.

Пример 2.12. PP2E\System\Filetools\commands.py

```
#!/usr/local/bin/python
from sys import argv
from scanfile import scanner
def processLine(line):                      # определить функцию,
    if line[0] == '*':                      # применяемую к каждой строчке
        print "Ms.", line[1:-1]
    elif line[0] == '+':
        print "Mr.", line[1:-1]             # отрезать первый и последний символы
    else:
        raise 'unknown command', line       # возбудить исключительную ситуацию
filename = 'data.txt'
if len(argv) == 2: filename = argv[1]       # аргумент командной строки с
scanner(filename, processLine)              # именем файла запускает сканер
```

Если по каким-то неясным причинам текстовый файл *hillbillies.txt* содержит следующие строки:

```
*Granny
```

```
+Jethro
*Elly-Mae
+"Uncle Jed"
```

то наш сценарий commands можно выполнить со следующим результатом:

```
C:\...\PP2E\System\Filetools>python commands.py hillbillies.txt
Ms. Granny
Mr. Jethro
Ms. Elly-Mae
Mr. "Uncle Jed"
```

На практике, однако, обычно можно добиться увеличения скорости обработки, перенеся ее из кода Python во встроенные инструменты. Например, если скорость важна (а нехватки памяти не испытывается), можно ускорить работу файлового сканера с помощью метода `readlines`, чтобы загрузить сразу весь файл в список, не прибегая к циклу `readline`:

```
def scanner(name, function):
    file = open(name? 'r')              # создать объект файла
    for line in file.readlines():       # получить сразу все строки
        function(line)                  # вызвать объект функции
    file.close()
```

А если есть список строк, еще больших чудес можно достичь с помощью встроенной функции `map`. Вот минималистская версия: цикл `for` заменен `map`, и Python сам закрывает файл при уборке мусора (или выходе из сценария):

```
def scanner(name, function):
    map(function, open(name, 'r').readlines())
```

Но как быть, если во время сканирования файла мы захотим его *изменить*? В примере 2.13 показаны два подхода: в одном используются явные файлы, а в другом стандартные потоки ввода/вывода, чтобы разрешить переадресацию в командной строке.

Пример 2.13. PP2E\System\Filetools\filters.py

```
def filter_files(name, function):         # фильтрация файла через функцию
    input  = open(name, 'r')              # создать явные объекты файлов
    output = open(name + '.out', 'w')     # и явный выходной файл
    for line in input.readlines():
        output.write(function(line))      # запись модифицированной строки
    input.close()
    output.close()                        # у выдачи суффикс '.out'

def filter_stream(function):
    import sys                            # без явных файлов
    while 1:                              # использовать стандартные потоки
        line = sys.stdin.readline()       # или: raw_input()
        if not line: break
        print function(line),             # или: sys.stdout.write()

if __name__ == '__main__':
    filter_stream(lambda line: line)      # копировать stdin в stdout, если выполнение
```

Поскольку стандартные потоки открываются без нашего участия, часто их проще использовать. Этот модуль более полезен при импорте в качестве библиотеки (клиент предоставляет функцию обработки строк); при автономном запуске просто копируется `stdin` в `stdout`:

```
C:\...\PP2E\System\Filetools>python filters.py < ..\System.txt
```

```
This directory contains operating system interface examples.

Many of the examples in this unit appear elsewhere in the examples distribution
tree, because they are actually used to manage other programs.
See the README.txt files in the subdirectories here for pointers.
```

Очень наблюдательные читатели могут заметить, что последний файл называется *filters.py* (с «s»), а не *filter.py*. Первоначально я назвал его именно как в последнем варианте, но изменил имя, заметив, что простой импорт имени файла (например, «import filter») присваивает модуль локальному имени «filter», из-за чего маскируется встроенная функция `filter`. Это встроенное средство функционального программирования, редко используемое в обычных сценариях; но стоит остерегаться выбора встроенных имен в качестве имен файлов модулей. Я, если позволите, буду.

Представление файлов как списков

Последний прием работы с файлами оказался настолько популярным, что заслужил быть представленным здесь. Хотя объекты файлов только экспортируют вызовы методов (например, `file.read()`), с помощью классов легко сделать их более похожими на структуры данных и скрыть некоторые детали используемых вызовов файлов. Модуль в примере 2.14 определяет объект `FileList`, служащий «оболочкой» реального файла, в которую добавлена поддержка последовательного индексирования.

Пример 2.14. PP2E\System\Filetools\filelist.py

```
class FileList:
    def __init__(self, filename):
        self.file = open(filename, 'r')     # открыть и сохранить файл
    def __getitem__(self, i):               # перегрузить индексирование
        line = self.file.readline()
        if line:
            return line                     # возвратить очередную строку
        else:
            raise IndexError                # для конца цикла 'for', 'in'
    def __getattr__(self, name):
        return getattr(self.file, name)     # другие атрибуты действительного файла
```

В этом классе определены три метода со специальными именами:

- Метод `__init__` вызывается при создании нового объекта.
- Метод `__getitem__` перехватывает операции индексации.
- Метод `__getattr__` обрабатывает ссылки на неопределенные атрибуты.

Этот класс в основном является лишь расширением встроенного объекта файла, в которое добавлен доступ по индексу. Большинство вызовов стандартных методов файла просто делегируется (передается) заключенному в оболочку файлу через `__getattr__`. При каждом обращении к объекту `FileList` по индексу его метод `__getitem__` возвращает очередную строчку реального файла. Поскольку циклы `for` действуют, повторно индексируя объекты, этот класс позволяет произвести итерацию заключенного в оболочку файла, как если бы это был находящийся в памяти список:

```
>>> from filelist import FileList
>>> for line in FileList('hillbillies.txt'):
...     print '>', line,
```

```
...
> *Granny
> +Jethro
> *Elly-Mae
> +¨Uncle Jed¨
```

Этот класс можно было бы сделать значительно более сложным и похожим на список. Например, можно перегрузить операцию +, чтобы она сцепляла файл с концом выходного файла, разрешить операции прямого индексирования, осуществляющие перемещение по строкам файла для получения заданного смещения, и т. д. Но поскольку кодирование всех таких расширений заняло бы больше места, чем здесь отпущено, они предлагаются читателю в качестве упражнения.

Средства для работы с каталогами

Одной из наиболее частых задач для утилит оболочки является применение операции к множеству файлов, находящихся в *каталоге* – «папке» на языке Windows. Выполняя сценарий с пакетом файлов, можно автоматизировать задачи, которые в противном случае пришлось бы многократно повторять вручную.

Допустим, например, что нужно найти во всех файлах Python из каталога разработки имя глобальной переменной (вы могли забыть, где оно используется). Есть много специфических для каждой платформы способов сделать это (например, команда *grep* в Unix), но сценарии Python, выполняющие такие задачи, будут работать на любой платформе, на которой работает Python – в Windows, Unix, Linux, Macintosh и практически любой, используемой на сегодняшний день. Просто скопируйте свой сценарий на любую машину, на которой вы хотите его использовать, и он будет работать независимо от имеющихся на ней утилит.

Обход одного каталога

Чаще всего при написании таких средств сначала получают список имен файлов, которые нужно обработать, а затем пошагово проходят этот список в цикле Python `for`, поочередно обрабатывая каждый файл. Весь фокус, следовательно, состоит в том, чтобы научиться получать такой список содержимого каталога в сценарии. Есть не менее трех способов сделать это: выполнить команды оболочки для получения списка через `os.popen`, найти файлы по шаблону имени с помощью `glob.glob` и получить перечень содержимого каталога с помощью `os.listdir`. Эти способы различаются по интерфейсу, формату результата и переносимости.

Запуск команд перечня каталога через os.popen

Ну-ка быстро скажите, как вы получали списки файлов в каталоге до того, как услышали про Python? Если у вас нет опыта программирования инструментов оболочки, ответ может быть следующим: «Ну, я запускал в Windows проводник и щелкал, куда нужно». Но здесь у нас идет речь о менее ориентированных на GUI механизмах командной строки (и ответы, использующие Perl и Tcl, лишь частично принимаются во внимание).

В Unix списки файлов в каталогах обычно получаются при вводе в оболочке команды *ls*; в Windows их можно создавать с помощью команды *dir*, вводимой в окне консоли MS-DOS. Поскольку сценарии Python могут выполнить любую команду оболочки с помощью `os.popen`, это самый общий способ получения содержимого каталога из программы Python. Мы уже встречались с `os.popen` ранее в этой главе: она выполняет стро-

ку с командой оболочки и возвращает файловый объект, из которого можно прочесть выдачу команды. Для иллюстрации допустим сначала, что есть такая структура каталога (да, на моем переносном компьютере с Windows есть обе команды – *dir* и *ls*; старые привычки умирают с трудом):

```
C:\temp>dir /B
about-pp.html
python1.5.tar.gz
about-pp2e.html
about-ppr2e.html
newdir

C:\temp>ls
about-pp.html     about-ppr2e.html  python1.5.tar.gz
about-pp2e.html   newdir

C:\temp>ls newdir
more   temp1  temp2  temp3
```

Имя *newdir* является здесь подкаталогом, вложенным в *C:\temp*. Итак, сценарии могут получать списки имен файлов и каталогов на этом уровне, просто порождая специфическую для платформы командную строку и читая полученную выдачу (текст, обычно выводимый в окно консоли):

```
C:\temp>python
>>> import os
>>> os.popen('dir /B').readlines()
['about-pp.html\012', 'python1.5.tar.gz\012', 'about-pp2e.html\012',
'about-ppr2e.html\012', 'newdir\012']
```

Возвращаемые командой оболочки строки содержат замыкающий символ конца строки, но его легко отрезать:

```
>>> for line in os.popen('dir /B').readlines():
...     print line[:-1]
...
about-pp.html
python1.5.tar.gz
about-pp2e.html
about-ppr2e.html
newdir
```

Обе команды *dir* и *ls* позволяют задавать шаблон имен файлов, которые нужно искать, и имена каталогов, список содержимого которых должен быть получен; опять же, мы просто выполняем здесь команды оболочки, поэтому годится все, что можно ввести в ответ на приглашение оболочки:

```
>>> os.popen('dir *.html /B').readlines()
['about-pp.html\012', 'about-pp2e.html\012', 'about-ppr2e.html\012']

>>> os.popen('ls *.html').readlines()
['about-pp.html\012', 'about-pp2e.html\012', 'about-ppr2e.html\012']

>>> os.popen('dir newdir /B').readlines()
['temp1\012', 'temp2\012', 'temp3\012', 'more\012']

>>> os.popen('ls newdir').readlines()
['more\012', 'temp1\012', 'temp2\012', 'temp3\012']
```

Эти вызовы используют общие средства и все действуют, как объявлено. Однако выше отмечалось, что недостатком `os.popen` является непереносимость (он не очень хоро-

шо работает в приложениях Windows GUI в Python 1.5.2 и более ранних и требует использования специфических для платформ команд), а кроме того, снижается производительность из-за необходимости запуска независимой программы. Следующие два альтернативных приема проявляют себя лучше в обоих отношениях.

Модуль glob

Термин «globbing» (глобальный поиск по шаблону) происходит от группового символа * в шаблонах имен файлов: на языке компьютерщиков * соответствует «glob» (комок) символов. Выражаясь менее поэтически, глобальный поиск по шаблону означает получение имен всех записей каталога – файлов и каталогов, – имена которых соответствуют заданному шаблону. В системах Unix при глобальном поиске шаблоны имен файлов в командной строке расширяются во все совпадающие имена еще перед выполнением команды. В Python можно делать нечто аналогичное, вызывая встроенную функцию `glob.glob` с шаблоном (возможности которого несколько расширены):

```
>>> import glob
>>> glob.glob('*')
['about-pp.html', 'python1.5.tar.gz', 'about-pp2e.html', 'about-ppr2e.html',
'newdir']

>>> glob.glob('*.html')
['about-pp.html', 'about-pp2e.html', 'about-ppr2e.html']

>>> glob.glob('newdir/*')
['newdir\\temp1', 'newdir\\temp2', 'newdir\\temp3', 'newdir\\more']
```

Вызов `glob` принимает обычный синтаксис шаблонов имен файлов, используемый оболочками (например, ? означает один любой символ, * означает любое число символов, а [] означает множество, из которого можно выбрать символ).[1] Если поиск нужно осуществлять в каталоге, отличном от текущего рабочего, в шаблон нужно включить путь к каталогу; разделитель имен каталогов может быть в стиле Unix или DOS (/ или \). Этот вызов также реализован без порождения команды оболочки и потому должен выполняться быстрее и лучше переноситься на все платформы Python, чем показанные выше схемы с `os.popen`.

Формально говоря, вызов `glob` несколько мощнее, чем здесь описано. Получение перечня файлов в каталоге является лишь одной из его возможностей поиска по каталогу. Например, его можно использовать для получения соответствующих имен из нескольких каталогов, так как каждый уровень в передаваемом пути к каталогу также может быть задан с помощью шаблона:

```
C:\temp>python
>>> import glob
>>> for name in glob.glob('*examples/L*.py'): print name
...
cpexamples\Launcher.py
cpexamples\Launch_PyGadgets.py
cpexamples\LaunchBrowser.py
cpexamples\launchmodes.py
examples\Launcher.py
examples\Launch_PyGadgets.py
examples\LaunchBrowser.py
```

[1] В действительности `glob` просто использует стандартный модуль `fnmatch` для поиска имен по шаблону; подробности смотрите в описании `fnmatch` далее в этой главе в разделе «Сделаем собственный модуль find».

```
examples\launchmodes.py

>>> for name in glob.glob(r'*\*\visitor_find*.py'): print name
...
cpexamples\PyTools\visitor_find.py
cpexamples\PyTools\visitor_find_quiet2.py
cpexamples\PyTools\visitor_find_quiet1.py
examples\PyTools\visitor_find.py
examples\PyTools\visitor_find_quiet2.py
examples\PyTools\visitor_find_quiet1.py
```

В первом вызове получены имена файлов из двух разных каталогов, соответствующих шаблону `*examples`; во втором примере оба первых уровня каталога заданы групповым символом, поэтому Python проверяет все возможные способы достичь основные имена файлов. Порождая команды оболочки с помощью `os.popen`, такого результата можно добиться, только если это умеет делать соответствующая оболочка или команда вывода перечня.

Вызов os.listdir

Вызов модуля `os listdir` предоставляет еще один способ составить из имен файлов список Python. Он принимает просто строку имени каталога, а не шаблон имени файла, и возвращает список, содержащий имена всех записей в каталоге – как просто файлов, так и вложенных каталогов, – для использования в вызывающем сценарии:

```
>>> os.listdir('.')
['about-pp.html', 'python1.5.tar.gz', 'about-pp2e.html', 'about-ppr2e.html',
'newdir']

>>> os.listdir(os.curdir)
['about-pp.html', 'python1.5.tar.gz', 'about-pp2e.html', 'about-ppr2e.html',
'newdir']

>>> os.listdir('newdir')
['temp1', 'temp2', 'temp3', 'more']
```

Это тоже делается, не прибегая к командам оболочки, и потому переносимо на все основные платформы Python. Результат не упорядочен никаким образом (но может быть отсортирован методом `sort`), возвращает базовые имена файлов без префиксов путей каталогов и содержит имена как файлов, так и каталогов для выводимого уровня.

Чтобы сравнить все три способа, запустим их рядом друг с другом для явно заданного каталога. В некоторых отношениях они различаются, но являются вариациями на одну и ту же тему: `os.popen` сортирует имена и возвращает символы конца строки, `glob.glob` принимает шаблоны и возвращает имена файлов с префиксами каталогов, а `os.listdir` принимает обычное имя каталога и возвращает имена файлов без префиксов каталога:

```
>>> os.popen('ls C:\PP2ndEd').readlines()
['README.txt\012', 'cdrom\012', 'chapters\012', 'etc\012', 'examples\012',
'examples.tar.gz\012', 'figures\012', 'shots\012']

>>> glob.glob('C:\PP2ndEd\*')
['C:\\PP2ndEd\\examples.tar.gz', 'C:\\PP2ndEd\\README.txt',
'C:\\PP2ndEd\\shots', 'C:\\PP2ndEd\\figures', 'C:\\PP2ndEd\\examples',
'C:\\PP2ndEd\\etc', 'C:\\PP2ndEd\\chapters', 'C:\\PP2ndEd\\cdrom']

>>> os.listdir('C:\PP2ndEd')
['examples.tar.gz', 'README.txt', 'shots', 'figures', 'examples', 'etc',
'chapters', 'cdrom']
```

Из всех трех способов обычно, если важна переносимость сценария, лучшими вариантами служат `glob` и `listdir`; при этом `listdir` в последних версиях Python, по-видимому, является самым быстрым (однако замерьте производительность сами: реализация может со временем измениться).

Расщепление и объединение результатов вывода

В последнем примере отмечалось, что `glob` возвращает имена с путями каталогов, а `listdir` дает необработанные базовые имена. В сценариях для удобства обработки часто требуется расщеплять результаты `glob`, чтобы получить базовые имена, либо расширять результаты `listdir` в полные пути. Такую трансляцию легко проводить, если позволить модулю `os.path` выполнить всю работу. Например, сценарию, который должен скопировать все файлы в какое-то место, обычно нужно сначала выделить базовые имена файлов из результатов, полученных `glob`, чтобы добавить впереди них другие имена каталогов:

```
>>> dirname = r'C:\PP2ndEd'
>>> for file in glob.glob(dirname + '/*'):
...     head, tail = os.path.split(file)
...     print head, tail, '=>', ('C:\\Other\\' + tail)
...
C:\PP2ndEd examples.tar.gz => C:\Other\examples.tar.gz
C:\PP2ndEd README.txt => C:\Other\README.txt
C:\PP2ndEd shots => C:\Other\shots
C:\PP2ndEd figures => C:\Other\figures
C:\PP2ndEd examples => C:\Other\examples
C:\PP2ndEd etc => C:\Other\etc
C:\PP2ndEd chapters => C:\Other\chapters
C:\PP2ndEd cdrom => C:\Other\cdrom
```

Здесь после `=>` показаны имена, в которые могут быть перемещены файлы. И наоборот, сценарию, который должен обработать все файлы в каталоге, отличном от того, в котором он выполняется, вероятно, потребуется предпослать результатам `listdir` имя целевого каталога, прежде чем предавать имена файлов другим инструментам:

```
>>> for file in os.listdir(dirname):
...     print os.path.join(dirname, file)
...
C:\PP2ndEd\examples.tar.gz
C:\PP2ndEd\README.txt
C:\PP2ndEd\shots
C:\PP2ndEd\figures
C:\PP2ndEd\examples
C:\PP2ndEd\etc
C:\PP2ndEd\chapters
C:\PP2ndEd\cdrom
```

Обход деревьев каталогов

Обратите, однако, внимание, что все предшествующие приемы возвращают имена файлов только из *одного* каталога. А что если нужно применить операцию к каждому файлу в каждом каталоге и подкаталоге дерева каталогов?

Допустим, например, снова, что требуется найти все вхождения глобального имени в сценариях Python. Однако на этот раз наши сценарии организованы в виде *пакета* модулей: каталога со вложенными подкаталогами, которые могут содержать собственные подкаталоги. Можно запустить наш гипотетический поисковый механизм

для одного каталога в каждом каталоге дерева вручную, но это утомительно, чревато ошибками и точно не доставит удовольствия.

К счастью, в Python обработать дерево каталогов почти так же просто, как и один каталог. Можно предварительно получить имена с помощью модуля `find`, написать рекурсивную процедуру для обхода дерева или использовать утилиту перемещения по дереву, встроенную в модуль `os`. Такие средства можно применять для поиска, копирования, сравнения, а также обрабатывать произвольные деревья каталогов другими способами на любой платформе, где выполняется Python (то есть почти всюду).

Модуль find

Первый способ действий в иерархии состоит в том, чтобы заранее собрать список всех имен в дереве каталогов в список и пройти по этому списку в цикле. Как и средства работы с одним каталогом, с которыми мы только что познакомились, вызов встроенной функции `find.find` возвращает список, состоящий из имен и файлов и каталогов. В отличие от описывавшихся средств, `find.find` возвращает также имена путей найденных файлов, содержащихся в подкаталогах, на всю глубину дерева:

```
C:\temp>python
>>> import find
>>> find.find('*')
['.\\about-pp.html', '.\\about-pp2e.html', '.\\about-ppr2e.html',
'.\\newdir', '.\\newdir\\more', '.\\newdir\\more\\xxx.txt',
'.\\newdir\\more\\yyy.txt', '.\\newdir\\temp1', '.\\newdir\\temp2',
'.\\newdir\\temp3', '.\\python1.5.tar.gz']

>>> for line in find.find('*'): print line
...
.\about-pp.html
.\about-pp2e.html
.\about-ppr2e.html
.\newdir
.\newdir\more
.\newdir\more\xxx.txt
.\newdir\more\yyy.txt
.\newdir\temp1
.\newdir\temp2
.\newdir\temp3
.\python1.5.tar.gz
```

Возвращается список полных путей, каждое из которых содержит путь к каталогу верхнего уровня. По умолчанию `find` ищет имена, соответствующие переданному шаблону, в дереве, корень которого находится в текущем рабочем каталоге, обозначаемом «.». Если нужен список, заданный более точно, можно передать как шаблон имени файла, так и корень дерева каталогов, с которого нужно начать. Вот как получить имена файлов HTML в «.» и ниже:

```
>>> find.find('*.html', '.')
['.\\about-pp.html', '.\\about-pp2e.html', '.\\about-ppr2e.html']
```

Случилось так, что `find.find` также является эквивалентом из библиотеки Python для команд, специфических для платформ, таких как *find –print* в Unix и Linux и *dir /B /S* в DOS и Windows. Поскольку обычно такие команды можно запустить из сценария Python с помощью `os.popen`, следующий код выполняет ту же работу, что и `find.find`, но ему свойственны непереносимость и необходимость запуска отдельной программы:

```
>>> import os
```

```
>>> for line in os.popen('dir /B /S').readlines(): print line,
...
C:\temp\about-pp.html
C:\temp\python1.5.tar.gz
C:\temp\about-pp2e.html
C:\temp\about-ppr2e.html
C:\temp\newdir
C:\temp\newdir\temp1
C:\temp\newdir\temp2
C:\temp\newdir\temp3
C:\temp\newdir\more
C:\temp\newdir\more\xxx.txt
C:\temp\newdir\more\yyy.txt
```

Если окажется, что вызов `find` не работает в вашей установке Python, попробуйте изменить утверждение, выполняющее импорт модуля, с `import find` на `from PP2E.PyTools import find`. Увы, модуль `find` стандартной библиотеки Python помечен в Python 1.6 как «устаревший». Это означает, что в будущем он может быть исключен из стандартного дистрибутива Python, поэтому обратите внимание на следующий раздел: мы воспользуемся позднее его материалом, чтобы написать собственный модуль `find` – его можно также найти на прилагаемом к книге CD.

Обходчик os.path.walk

Чтобы облегчить применение операции ко всем файлам дерева, с Python поставляется утилита, сканирующая деревья и запускающая в каждом каталоге указанную функцию. Функция `os.path.walk` вызывается с корнем каталога, объектом функции и необязательным элементом данных и обходит дерево от корня каталога и ниже. В каждом каталоге переданный объект функции вызывается с необязательным элементом данных, именем текущего каталога и списком имен файлов в этом каталоге (полученном `os.listdir`). Обычно передаваемая функция просматривает список имен файлов, чтобы обработать файлы на каждом уровне каталогов дерева.

Увидев это описание в первый раз, может показаться, что оно ужасно сложное, но после привыкания к `os.path.walk` все окажется довольно простым. В следующем коде, например, функция `lister` вызывается из `os.path.walk` в каждом каталоге, корень дерева которого находится в «.». Попутно `lister` просто выводит имя каталога и все файлы на текущем уровне (предварив их именем каталога). Описать это на Python проще, чем на обычном языке:

```
>>> import os
>>> def lister(dummy, dirname, filesindir):
...     print '[' + dirname + ']'
...     for fname in filesindir:
...         print os.path.join(dirname, fname)         # обработать один файл
...
>>> os.path.walk('.', lister, None)
[.]
.\about-pp.html
.\python1.5.tar.gz
.\about-pp2e.html
.\about-ppr2e.html
.\newdir
```

```
[.\newdir]
.\newdir\temp1
.\newdir\temp2
.\newdir\temp3
.\newdir\more
[.\newdir\more]
.\newdir\more\xxx.txt
.\newdir\more\yyy.txt
```

Иными словами, мы запрограммировали собственное, легко изменяемое, средство вывода каталога в Python. Поскольку нам может потребоваться изменять его и всюду использовать, сделаем его постоянно доступным с помощью файла модуля, показанного в примере 2.15, после того как детали разработаны интерактивно.

Пример 2.15. PP2E\System\Filetools\lister_walk.py

```
# перечислить дерево файлов с помощью os.path.walk
import sys, os
def lister(dummy, dirName, filesInDir):          # вызывается с каждым каталогом
    print '[' + dirName + ']'
    for fname in filesInDir:                     # включает имена подкаталогов
        path = os.path.join(dirName, fname)      # добавить префикс имени с каталогом
        if not os.path.isdir(path):              # вывести только простые файлы
            print path
if __name__ == '__main__':
    os.path.walk(sys.argv[1], lister, None)      # имя каталога в командной строке
```

Этот тот же самый код, за исключением того, что имена каталогов отфильтровываются из списка имен файлов с помощью проверки `os.path.isdir`, чтобы избежать их повторного вывода (вот он уже и изменен). При таком оформлении код можно также выполнять из командной строки оболочки: вот пример запуска его из другого каталога с передачей в аргументе командной строки каталога, список содержимого которого должен быть выведен:

```
C:\...\PP2E\System\Filetools>python lister_walk.py C:\Temp
[C:\Temp]
C:\Temp\about-pp.html
C:\Temp\python1.5.tar.gz
C:\Temp\about-pp2e.html
C:\Temp\about-ppr2e.html
[C:\Temp\newdir]
C:\Temp\newdir\temp1
C:\Temp\newdir\temp2
C:\Temp\newdir\temp3
[C:\Temp\newdir\more]
C:\Temp\newdir\more\xxx.txt
C:\Temp\newdir\more\yyy.txt
```

Парадигма `walk` позволяет также функциям перекраивать набор посещенных каталогов, изменяя аргумент списка файлов по месту. В руководстве по библиотеке рассказывается об этом подробнее, но, вероятно, более поучительно узнать, как в действительности выглядит `walk`. Вот реальная, написанная на Python реализация для платформ Windows, которая дополнена комментариями, призванными лишить ее действия ореола таинственности:

```
def walk(top, func, arg):                    # top - имя текущего каталога
    try:
        names = os.listdir(top)              # получить все имена файлов/каталогов
```

```
    except os.error:                        # у них нет префикса пути
        return
    func(arg, top, names)                   # запуск func со списком имен
    exceptions = ('.', '..')
    for name in names:                      # пройти тот же список,
        if name not in exceptions:          # пропустив себя/родителя
            name = join(top, name)          # добавить к имени префикс пути
            if isdir(name):
                walk(name, func, arg)       # войти в подкаталоги
```

Обратите внимание, что `walk` создает на каждом уровне списки имен с помощью `os.listdir`, – вызова, возвращающего имена файлов и каталогов без какого-либо порядка и без путей к каталогам. Заметьте также, что `walk` использует список, возвращаемый `os.listdir` и передаваемый в заданную функцию, чтобы затем спуститься в подкаталоги (переменная `names`). Так как списки являются модифицируемыми объектами, которые можно изменять по месту, то изменения, производимые вашей функцией в переданном ей списке имен, оказывают воздействие на то, что будет `walk` делать дальше. Например, удаление имен каталогов обрежет ветви обхода, а сортировка списка упорядочит движение.

Рекурсивный обход os.listdir

Средство `os.path.walk` осуществляет обход дерева за нас, но иногда большей гибкости можно достичь самостоятельно, едва ли потратив больше труда. В следующем сценарии представлен другой код для вывода содержимого каталога с использованием рекурсивной функции обхода вручную. Функция `mylister` в примере 2.16 почти такая же, как `lister` в предыдущем сценарии, но вызывает `os.listdir`, чтобы создать пути к файлам вручную, и вызывает себя рекурсивно, чтобы спуститься в подкаталоги.

Пример 2.16. PP2E\System\Filetools\lister_recur.py

```
# вывод перечня файлов в дереве каталогов с помощью рекурсии
import sys, os
def mylister(currdir):
    print '[' + currdir + ']'
    for file in os.listdir(currdir):              # здесь получение списка файлов
        path = os.path.join(currdir, file)        # добавить путь к каталогу
        if not os.path.isdir(path):
            print path
        else:
            mylister(path)                        # рекурсия в подкаталоги
if __name__ == '__main__':
    mylister(sys.argv[1])                         # имя каталога в командной строке
```

Эта версия тоже оформлена в виде сценария (кода явно слишком много, чтобы вводить его в интерактивном приглашении); при запуске в качестве сценария получается идентичная выдача:

```
C:\...\PP2E\System\Filetools>python lister_recur.py C:\Temp
[C:\Temp]
C:\Temp\about-pp.html
C:\Temp\python1.5.tar.gz
C:\Temp\about-pp2e.html
C:\Temp\about-ppr2e.html
[C:\Temp\newdir]
C:\Temp\newdir\temp1
C:\Temp\newdir\temp2
C:\Temp\newdir\temp3
```

```
[C:\Temp\newdir\more]
C:\Temp\newdir\more\xxx.txt
C:\Temp\newdir\more\yyy.txt
```

Но этот файл столь же полезен при импорте и вызове из любого места:

```
C:\temp>python
>>> from PP2E.System.Filetools.lister_recur import mylister
>>> mylister('.')
[.]
.\about-pp.html
.\python1.5.tar.gz
.\about-pp2e.html
.\about-ppr2e.html
[.\newdir]
.\newdir\temp1
.\newdir\temp2
.\newdir\temp3
[.\newdir\more]
.\newdir\more\xxx.txt
.\newdir\more\yyy.txt
```

Мы еще воспользуемся большей частью приемов, приведенных в этом разделе, в главе 5 и книге в целом. Например, сценарии для копирования и сравнения деревьев каталогов используют приведенные выше технологии обхода деревьев. По ходу изложения вы увидите эти инструменты в действии. Если вас интересует обработка каталогов, посмотрите также описание старого модуля Python `grep` в главе 5: он ищет файлы и может применяться ко всем файлам в каталоге, если сочетается с модулем `glob`, но сам по себе просто выводит результаты и не обходит деревья каталогов.

Сделаем собственный модуль find

За последние восемь лет я стал доверять Великодушному Диктатору Python. Гвидо обычно делает правильные вещи, и если вы так не считаете, то только потому, что еще не осознали ошибочности своей позиции. Поверьте мне. С другой стороны, не вполне ясно, почему стандартный модуль `find`, о котором я вам рассказывал, впал в немилость: это полезный инструмент. На практике я много использую его: часто удобно получить список файлов для обработки за один вызов функции и пройтись по нему в цикле `for`. Альтернативные способы – `os.path.walk` и рекурсивные функции – требуют большего объема кодирования и труднее усваиваются новичками.

Я полагаю, что поклонники модуля `find` (если такие есть) могли бы защитить его в результате долгих дебатов в Интернете, которые затянулись бы на недели и привлекли многочисленных героических участников, ничего не добившихся бы в итоге. Я решил потратить десять минут и состряпать собственный альтернативный вариант. Модуль примера 2.17 использует стандартный вызов `os.path.walk`, описанный выше, чтобы заново реализовать операцию поиска для Python.

Пример 2.17. PP2E\PyTools\find.py

```
#!/usr/bin/python
################################################################################
# пользовательская версия модуля find, объявленного устаревшим в стандартной библиотеке –
# импортируется как "PyTools.find"; эквивалентен оригиналу, но использует os.path.walk,
# не поддерживает обрезания подкаталогов в дереве и настроен так, чтобы выполняться
# как сценарий верхнего уровня; сортировка результирующего списка несколько отличается
# для разных деревьев; использует распаковку выборки в списках аргументов функций;
################################################################################
```

```
import fnmatch, os
def find(pattern, startdir=os.curdir):
    matches = []
    os.path.walk(startdir, findvisitor, (matches, pattern))
    matches.sort()
    return matches
def findvisitor((matches, pattern), thisdir, nameshere):
    for name in nameshere:
        if fnmatch.fnmatch(name, pattern):
            fullpath = os.path.join(thisdir, name)
            matches.append(fullpath)
if __name__ == '__main__':
    import sys
    namepattern, startdir = sys.argv[1], sys.argv[2]
    for name in find(namepattern, startdir): print name
```

В этом файле нет ничего особенного, но вызов его функции `find` предоставляет такую же утилиту, как устаревший стандартный модуль `find`, и это существенно проще, чем переписывать заново весь код этого файла каждый раз, когда требуется выполнить поиск типа find. Например, для обработки всех файлов Python в дереве я просто ввожу:

```
from PP2E.PyTools import find
for name in find.find('*.py'):
    ...и делаю что-нибудь с name...
```

Более конкретный пример дает следующий простой сценарий, которым я пользуюсь для удаления старых текстовых выходных файлов, находящихся где-либо в дереве примеров книги:

```
C:\...\PP2E>type PyTools\cleanoutput.py
import os                                   # удалить старые выходные файлы в дереве
from PP2E.PyTools.find import find          # полный путь нужен, только если модуль перемещен
for filename in find('*.out.txt'):          # в Linux вместо type использовать cat
    print filename
    if raw_input('View?') == 'y':
        os.system('type ' + filename)
    if raw_input('Delete?') == 'y':
        os.remove(filename)
C:\temp\examples>python %X%\PyTools\cleanoutput.py
.\Internet\Cgi-Web\Basics\languages.out.txt
View?
Delete?
.\Internet\Cgi-Web\PyErrata\AdminTools\dbaseindexed.out.txt
View?
Delete?y
```

Чтобы получить такой экономичный код, пользовательский модуль `find` вызывает `os.path.walk` для регистрации функции, которая должна вызываться с каждым каталогом в дереве, и просто добавляет попутно в результирующий список все подходящие имена файлов.

Здесь, однако, используется новый модуль `fnmatch` – стандартный модуль Python, выполняющий поиск имен по шаблону в стиле Unix, использовавшийся также в первоначальном `find`. Этот модуль поддерживает обычные операторы в строках шаблонов имен: `*` (соответствует любому числу символов), `?` (соответствует любому одиночному символу) и `[...]` и `[!...]` (соответствуют любому символу, заключенному в пару скобок, или отсутствующему в ней); прочие символы соответствуют самим себе.[1] Чтобы убедиться в том, что этот альтернативный способ дает аналогичные результаты, я написал тестовый модуль, показанный в примере 2.18.

Пример 2.18. PP2E\PyTools\find-test.py

```
##########################################################################
# проверка пользовательского find; встроенный модуль find устарел:
# если он когда-либо исчезнет вообще, замените все "import find"
# на "from PP2E.PyTools import find" (или добавьте PP2E\PyTools в значение
# пути, и тогда просто "import find"); этот сценарий за 4 секунды выполняет
# 10 поисков в каталоге из примерно 1500 имен на ноутбуке 650 МГц под Win98;
##########################################################################
import sys, os, string
for dir in sys.path:
    if string.find(os.path.abspath(dir), 'PyTools') != -1:
        print 'removing', repr(dir)
        sys.path.remove(dir)           # иначе могут импортироваться оба find из PyTools, '.'!

import find                            # получить устаревший встроенный (пока)
import PP2E.PyTools.find               # позднее так: from PP2E.PyTools import find
print  find
print  PP2E.PyTools.find

assert find.find != PP2E.PyTools.find.find         # действительно разные?
assert string.find(str(find), 'Lib') != -1         # должно быть после удаления пути
assert string.find(str(PP2E.PyTools.find), 'PyTools') != -1

startdir = r'C:\PP2ndEd\examples\PP2E'
for pattern in ('*.py', '*.html', '*.c', '*.cgi', '*'):
    print pattern, '=>'
    list1 = find.find(pattern, startdir)
    list2 = PP2E.PyTools.find.find(pattern, startdir)
    print len(list1), list1[-1]
    print len(list2), list2[-1]
    print list1 == list2,; list1.sort(); print list1 == list2
```

В начале этого сценария есть таинственный код, который нужно объяснить. Чтобы обеспечить загрузку как стандартного библиотечного модуля `find`, так и пользовательского из *PP2E\PyTools*, необходимо удалить записи в пути поиска модулей, указывающие на каталог *PP2E\PyTools*, и импортировать пользовательскую версию, полностью указав каталог пакета – `PP2E.PyTools.find`. Если этого не сделать, всегда будет использоваться один и тот же модуль `find` из *PyTools*, откуда бы ни запускался сценарий.

И вот почему. Вспомните, что Python всегда добавляет каталог, содержащий выполняемый сценарий, в *начало* `sys.path`. Если не удалить эту запись, утверждение `import find` всегда будет загружать пользовательский `find` из `PyTools`, потому что пользовательский модуль *find.py* находится в том же каталоге, что и сценарий *find-test.py*. В результате исходный каталог сценария будет маскировать стандартный библиотечный `find`. Если это не понятно, вернитесь назад и снова прочтите раздел «Текущий рабочий каталог» в начале этой главы.

Ниже приводятся вывод этого тестового сценария и несколько вызовов из командной строки: в отличие от первоначального `find`, пользовательскую версию примера 2.18 можно запускать и как инструмент командной строки. Если внимательно изучить выдачу теста, то можно заметить, что пользовательская версия отличается только случайным порядком сортировки, в который я не стану здесь вдаваться дальше (исход-

[1] В отличие от модуля `re`, `fnmatch` поддерживает только обычные операторы соответствия оболочки Unix, а не полноценные шаблоны регулярных выражений; смотрите подробнее в главе 18 о том, что это означает.

ный модуль `find` использовал рекурсивную функцию, а не `os.path.walk`); строки «0 1» означают, что результаты разнятся порядком, но не содержимым. Поскольку при вызове `find` обычно не требуется точного порядка получаемых имен файлов, это несущественно:

```
C:\temp>python %X%\PyTools\find-test.py
removing 'C:\\PP2ndEd\\examples\\PP2E\\PyTools'
<module 'find' from 'C:\Program Files\Python\Lib\find.pyc'>
<module 'PP2E.PyTools.find' from 'C:\PP2ndEd\examples\PP2E\PyTools\find.pyc'>
*.py =>
657 C:\PP2ndEd\examples\PP2E\tounix.py
657 C:\PP2ndEd\examples\PP2E\tounix.py
0 1
*.html =>
37 C:\PP2ndEd\examples\PP2E\System\Filetools\template.html
37 C:\PP2ndEd\examples\PP2E\System\Filetools\template.html
1 1
*.c =>
46 C:\PP2ndEd\examples\PP2E\Other\old-Integ\embed.c
46 C:\PP2ndEd\examples\PP2E\Other\old-Integ\embed.c
0 1
*.cgi =>
24 C:\PP2ndEd\examples\PP2E\Internet\Cgi-Web\PyMailCgi\onViewSubmit.cgi
24 C:\PP2ndEd\examples\PP2E\Internet\Cgi-Web\PyMailCgi\onViewSubmit.cgi
1 1
* =>
1519 C:\PP2ndEd\examples\PP2E\xferall.linux.csh
1519 C:\PP2ndEd\examples\PP2E\xferall.linux.csh
0 1
C:\temp>python %X%\PyTools\find.py *.cxx C:\PP2ndEd\examples\PP2E
C:\PP2ndEd\examples\PP2E\Extend\Swig\Shadow\main.cxx
C:\PP2ndEd\examples\PP2E\Extend\Swig\Shadow\number.cxx
C:\temp>python %X%\PyTools\find.py *.asp C:\PP2ndEd\examples\PP2E
C:\PP2ndEd\examples\PP2E\Internet\Other\asp-py.asp
C:\temp>python %X%\PyTools\find.py *.i C:\PP2ndEd\examples\PP2E
C:\PP2ndEd\examples\PP2E\Extend\Swig\Environ\environ.i
C:\PP2ndEd\examples\PP2E\Extend\Swig\Shadow\number.i
C:\PP2ndEd\examples\PP2E\Extend\Swig\hellolib.i
C:\temp>python %X%\PyTools\find.py setup*.csh C:\PP2ndEd\examples\PP2E
C:\PP2ndEd\examples\PP2E\Config\setup-pp-embed.csh
C:\PP2ndEd\examples\PP2E\Config\setup-pp.csh
C:\PP2ndEd\examples\PP2E\EmbExt\Exports\ClassAndMod\setup-class.csh
C:\PP2ndEd\examples\PP2E\Extend\Swig\setup-swig.csh
 [схема сортировки имен файлов]
C:\temp> python
>>> l = ['ccc', 'bbb', 'aaa', 'aaa.xxx', 'aaa.yyy', 'aaa.xxx.nnn']
>>> l.sort()
>>> l
['aaa', 'aaa.xxx', 'aaa.xxx.nnn', 'aaa.yyy', 'bbb', 'ccc']
```

Наконец, если в будущих версиях Python приведенный пример будет невозможно выполнить из-за отсутствия `find`, просто измените директиву импорта модуля `find` в исходном коде с `import find` на `from PP2E.PyTools import find`. В этом случае будет найден пользовательский модуль `find` в дереве каталогов примеров книги; старый модуль в стандартной библиотеке Python будет проигнорирован (если он там еще есть). А если вы наберетесь смелости добавить сам каталог *PP2E\PyTools* в значение PYTHONPATH, все первоначальные утверждения `import find` будут работать по-прежнему.

Еще лучше вообще ничего не делать: большинство примеров данной книги, использующих `find`, автоматически используют альтернативный вариант, перехватывая исключительную ситуацию, генерируемую при импорте, если они размещаются не в каталоге *PyTools*:

```
try:
    import find
except ImportError:
    from PP2E.PyTools import find
```

Модуль `find` может уйти, но забывать о нем не нужно.

Python и csh

Если вам знакомы другие распространенные языки сценариев оболочки, может оказаться полезным сравнение их с Python. Вот простой сценарий на языке оболочки Unix, называемой `csh`, который отправляет все файлы из текущего рабочего каталога с суффиксом *.py* (то есть все файлы исходного кода Python) по некоторому фиктивному адресу электронной почты:

```
#!/bin/csh
foreach x (*.py)
    echo $x
    mail eric@halfabee.com -s $x < $x
end
```

Эквивалентный сценарий Python выглядит сходным образом:

```
#!/usr/bin/python
import os, glob
for x in glob.glob('*.py'):
    print x
    os.system('mail eric@halfabee.com -s %s < %s' % (x, x))
```

но несколько более многословен. Так как Python, в отличие от `csh`, предназначен не только для использования в сценариях оболочки, системные интерфейсы должны импортироваться и вызываться в явном виде. А так как Python служит не просто языком для обработки строк, строки символов должны заключаться в кавычки, как в C.

Несмотря на то что в таких простых сценариях может потребоваться несколько дополнительных нажатий на клавиши, Python оказывается лучшим инструментом как только мы выходим за рамки тривиальных программ, благодаря тому что является языком общего назначения. Например, можно расширить предшествующий пример с тем, чтобы отправлять файлы по FTP, предоставлять через GUI возможность отбора сообщений, получать сообщения из базы данных SQL и применять объекты COM в Windows, и все это с использованием стандартных инструментов Python.

Сценарии Python обычно также оказываются лучше переносимыми на другие платформы, чем `csh`. Например, при использовании в Python для отправки электронной почты интерфейса SMTP вместо почтовых инструментов командной строки в Unix сценарий может выполняться на любой машине, где есть Python и соединение с Интернетом (как будет показано в главе 11 «Сценарии на стороне клиента», для SMTP требуются только сокеты). При этом, как и в C, не требуется `$` для вычисления значений переменных: чего еще можно пожелать от бесплатно распространяемого языка?

3

Системные средства параллельного выполнения

«Расскажите обезьянам, что им делать»

Большинство компьютеров проводит массу времени, ничего не делая. Если запустить системный монитор и посмотреть на коэффициент загрузки ЦП, вы поймете, что я имею в виду: редко можно увидеть, чтобы он достиг 100%, даже если выполняется несколько программ одновременно.[1] Просто в программном обеспечении существует очень много задержек – доступ к диску, сетевой трафик, запросы к базам данных, ожидание нажатия пользователем кнопки и тому подобное. На практике большая часть мощности современных ЦП часто не используется: более быстрые микросхемы дают ускорение во время пиков потребности в производительности, но значительная часть их мощности может в целом оказаться невостребованной.

Уже в начале эры вычислений программисты поняли, что могут воспользоваться такой неиспользуемой вычислительной мощностью, выполняя одновременно несколько программ. Если распределить внимание ЦП среди ряда задач, его мощность не будет тратиться впустую, пока некоторая конкретная задача ждет осуществления внешнего события. Такая технология обычно называется *параллельной обработкой*, потому что возникает впечатление одновременного выполнения заданий параллельно во времени. Это одна из центральных идей современных операционных систем, на основе которой возникло представление о компьютерных интерфейсах с несколькими активными окнами, воспринимаемое нами теперь, как само собой разумеющееся. Даже внутри одной задачи разделение обработки на ряд параллельно выполняющихся заданий может увеличить быстродействие системы в целом, во всяком случае по меркам внешних часов.

Столь же важно для современных систем обладание быстрой реакцией на действия пользователя независимо от объема работы, проводимой невидимо для него. Обычно недопустимо, чтобы при выполнении запроса программа зависала. Взгляните, например, на пользовательский интерфейс броузера электронной почты: обрабатывая запрос на получение почты с сервера, программа должна загрузить текст с сервера через сеть. Если почты достаточно много, а соединение с Интернетом достаточно медленное, для завершения этого этапа может потребоваться несколько минут. Но по ходу выполнения задачи загрузки программа в целом не должна остановиться: она по-прежнему должна реагировать на запрос обновления экрана, щелчок мыши и т. д.

---

[1] Под Windows нужно щелкнуть по кнопке Start, выбрать Programs/Accessories/System Tools/System Monitor и посмотреть на Processor Usage. Когда я писал этот текст, на моем портативном компьютере график редко подымался выше 50% (по крайней мере, пока я не ввел `while 1: pass` в окне консоли интерактивного сеанса Python).

И здесь на помощь приходит параллельная обработка. Выполняя такие долгоиграющие задачи параллельно с остальной частью программы, система в целом может сохранить способность реагировать на события независимо от того, насколько занятыми оказываются другие ее части.

Существует два встроенных способа одновременного выполнения задач в Python: *ветвление* (*forks*) процессов и порожденные *потоки* (*threads*). Функционально тот и другой способы используют сервисы соответствующей операционной системы, чтобы параллельно выполнять участки кода Python. Процедурно они очень различаются в отношении интерфейсов, переносимости и связи между задачами. На момент написания данной книги ветвление процессов не поддерживалось Windows (подробнее об этом сказано в замечании ниже), но поддержка потоков Python осуществляется на всех основных платформах. Кроме того, существуют иные специфические для Windows способы запуска программ, аналогичные ветвлению.

В данной главе, в которой продолжено рассмотрение системных интерфейсов, доступных программистам Python, будут исследованы встроенные средства Python для параллельного запуска программ, а также обмена информацией с этими программами. В некотором смысле мы приступили к этому раньше: вызовы `os.system` и `os.popen`, которые мы изучили и использовали в предыдущей главе, служат и весьма переносимым способом порождения программ командной строки и общения с ними. Теперь мы сделаем упор на знакомстве с более непосредственными приемами – ветвлением, потоками, каналами, сигналами и специфическими для Windows средствами запуска кода на выполнение. В следующей главе (и в оставшейся части книги) мы будем использовать эти приемы в приближенных к реальности программах, поэтому прежде чем двигаться вперед, необходимо усвоить основы.

Ветвление процессов

Ветвление процессов является традиционным способом организации параллельных вычислений и представляет собой фундаментальную часть инструментального набора Unix. Ветвление основано на понятии копирования (*copying*) программ: когда программа вызывает процедуру ветвления, операционная система создает в памяти новый экземпляр (копию) этой программы и запускает его параллельно оригиналу. В некоторых системах исходная программа в действительности не копируется (это дорогостоящая операция), но новый экземпляр работает так, как если бы он был подлинной копией.

После операции ветвления исходный экземпляр программы называется *родительским* процессом, а создаваемая с помощью `os.fork` копия называется *дочерним* процессом. Вообще говоря, родитель может произвести любое число потомков, а потомки могут создавать собственные дочерние процессы – все ответвленные процессы выполняются независимо и параллельно под управлением операционной системы. Возможно, это проще выглядит на практике, чем в теории; сценарий Python в примере 3.1 ответвляет новые дочерние процессы до нажатия в консоли «q».

Пример 3.1. PP2E\System\Processes\fork1.py

```
# ответвляет дочерние процессы до нажатия 'q'

import os

def child():
    print 'Hello from child',  os.getpid()
    os._exit(0)  # или возвращается в родительский цикл
```

```
def parent():
    while 1:
        newpid = os.fork()
        if newpid == 0:
            child()
        else:
            print 'Hello from parent', os.getpid(), newpid
        if raw_input() == 'q': break
parent()
```

Средства ветвления в Python, находящиеся в модуле os, служат просто тонкими оболочками вокруг стандартных средств ветвления из библиотеки C. Для запуска нового параллельного процесса вызовите встроенную функцию os.fork. Поскольку эта функция создает копию вызывающей программы, она возвращает различные значения в каждой копии: ноль в дочернем процессе и ID процесса нового потомка в родительской программе. Обычно программы проверяют этот результат, чтобы в дочернем процессе производить обработку иначе; например, в этом сценарии функция child выполняется только в дочерних процессах.[1]

К сожалению, в Windows это сегодня не работает: fork не стыкуется с моделью Windows и перенос этого вызова все еще находится в разработке. Но поскольку ветвление входит в состав модели программирования Unix, следующий сценарий хорошо работает в Unix и Linux:

```
[mark@toy]$ python fork1.py
Hello from parent 671 672
Hello from child 672

Hello from parent 671 673
Hello from child 673

Hello from parent 671 674
Hello from child 674
q
```

Эти сообщения представляют три ответвленных дочерних процесса; уникальные идентификаторы всех участвующих процессов получены и выведены с помощью вызова os.getpid. Тонкое место: функция процесса child явно завершает выполнение с помощью вызова os._exit. Этот вызов мы более подробно обсудим далее в этой главе, но если его не сделать, дочерний процесс продолжит существование после возврата из функции child (вспомните, что это лишь копия исходного процесса). В результате дочерний процесс возвратится в цикл в parent и начнет плодить собственных потомков (то есть у родителя появятся внуки). Если удалить вызов выхода и осуществить повторный запуск, то может понадобиться ввести для остановки несколько «q», поскольку в функции parent выполняется несколько процессов.

В примере 3.1 каждый процесс завершается вскоре после запуска, поэтому перекрытие по времени незначительно. Попробуем сделать нечто более сложное, чтобы лучше продемонстрировать параллельное выполнение нескольких ответвленных процессов. Пример 3.2 запускает 10 экземпляров себя самого, при этом каждый экземпляр считает до 10 с односекундной задержкой между итерациями. Встроенный вызов time.sleep

[1] По крайней мере в текущей реализации Python вызов os.fork в сценарии фактически копирует процесс интерпретатора Python (если взглянуть на список процессов, то после ветвления можно найти две записи Python). Но поскольку интерпретатор Python регистрирует все, касающееся выполнения вами сценария, можно считать fork непосредственным копированием вашей программы. Так и будет, если сценарии Python когда-нибудь станут компилироваться в двоичный машинный код.

просто задерживает вызывающий процесс на несколько секунд (задайте значение с плавающей точкой для остановки на дробную часть секунд).

Пример 3.2. PP2E\System\Processes\fork-count.py

```
##########################################################################################
# основы ветвления: запустить 10 экземпляров этой программы параллельно оригиналу;
# каждый экземпляр считает до 10 в том же потоке stdout – при ветвлении копируется
# память процесса, в том числе дескрипторы файлов; в настоящее время ветвление
# не действует в Windows: запускайте программы в Windows с помощью os.spawnv;
# примерно соответствует комбинации fork+exec;
##########################################################################################
import os, time
def counter(count):
    for i in range(count):
        time.sleep(1)
        print '[%s] => %s' % (os.getpid(), i)

for i in range(10):
    pid = os.fork()
    if pid != 0:
        print 'Process %d spawned' % pid
    else:
        counter(10)
        os._exit(0)

print 'Main process exiting.'
```

При прогоне этот сценарий сразу запускает 10 процессов и завершает работу. Все 10 ответвленных процессов оставляют первое показание счетчика секундой позже, а также каждую последующую секунду. Дочерние процессы продолжают выполняться, даже если создавший их родительский процесс прекратил существование:

```
mark@toy]$ python fork-count.py
Process 846 spawned
Process 847 spawned
Process 848 spawned
Process 849 spawned
Process 850 spawned
Process 851 spawned
Process 852 spawned
Process 853 spawned
Process 854 spawned
Process 855 spawned
Main process exiting.
[mark@toy]$
[846] => 0
[847] => 0
[848] => 0
[849] => 0
[850] => 0
[851] => 0
[852] => 0
[853] => 0
[854] => 0
[855] => 0
[847] => 1
[846] => 1
... остальная выдача опущена ...
```

Выдача всех этих процессов показывается на одном и том же экране, потому что все они использую один и тот же стандартный выходной поток. Технически ответвленный процесс получает копию глобальной памяти исходного процесса, в том числе дескрипторы открытых файлов. Из-за этого глобальные объекты типа файлов начинают работу в дочернем процессе с теми же значениями. Но важно помнить, что глобальная память копируется, а не используется совместно: если дочерний процесс изменит глобальный объект, то изменит только свой экземпляр этого объекта. (Как мы увидим, в потоках это не так, что будет темой следующего раздела.)

Комбинация fork/exec

В примерах 3.1 и 3.2 дочерние процессы просто запускали функцию в программе Python и завершали свою работу. В Unix-подобных платформах ветвление часто служит основой для запуска независимо выполняемых программ, совершенно отличных от программы, выполнившей вызов `fork`. Так, в примере 3.3 новые процессы ответвляются, пока не будет снова нажата клавиша «q», но в дочерних процессах запускается совершенно новая программа, а не вызывается функция из того же файла.

Пример 3.3. PP2E\System\Processes\fork-exec.py

```
# запускает программы, пока не будет введено 'q'

import os

parm = 0
while 1:
    parm = parm+1
    pid = os.fork()
    if pid == 0:                                                # копировать процесс
        os.execlp('python', 'python', 'child.py', str(parm)) # перегрузить программу
        assert 0, 'error starting program'                      # возврата быть не должно
    else:
        print 'Child is', pid
        if raw_input() == 'q': break
```

Если вы достаточно много занимались разработками для Unix, комбинация `fork/exec` может быть вам знакома. Главное, на что нужно обратить внимание в этом коде, это вызов `os.execlp`. Если сказать кратко, этот вызов перекрывает (*overlays*), то есть заменяет другой, выполняющуюся в данном процессе программу. Поэтому *комбинация* `os.fork` и `os.execlp` означает запуск нового процесса и выполнение в этом процессе новой программы, то есть, иными словами, запуск новой программы параллельно исходной.

Форматы вызова os.exec

Аргументы `os.execlp` задают программу, которая должна быть выполнена, с помощью аргументов командной строки, необходимых для запуска этой программы (то есть того, что в сценариях Python представлено как `sys.argv`). В случае успеха начинается выполнение новой программы, и возврата из самого вызова `os.execlp` не происходит (так как исходная программа заменена, то возвращаться действительно некуда). Если происходит возврат из вызова, значит, произошла ошибка, поэтому в коде после него стоит утверждение `assert`, при достижении которого всегда возбуждается исключительная ситуация.

В стандартной библиотеке Python есть несколько разновидностей `os.exec`; некоторые из них позволяют настраивать для новой программы переменные окружения, передавать в различном формате аргументы командной строки и т. д. Все они имеются как в Unix, так и в Windows и заменяют вызвавшую их программу (т. е. интерпретатор Py-

thon). Есть восемь разновидностей exec, что может несколько запутать, если не сделать обобщения:

os.execv(*program, commandlinesequence*)

В базовый формат «v»-вызова exec передается имя выполняемой программы вместе со списком или кортежем строк аргументов командной строки, используемых при запуске выполняемого модуля (то есть слов, которые обычно можно ввести в оболочке для запуска программы).

os.execl(*program, cmdarg1, cmdarg2, ... cmdargN*)

В базовый формат «l»-вызова exec передается имя выполняемой программы, за которым следуют один или более аргументов командной строки, передаваемых как отдельные аргументы функции. Это то же самое, что os.execv(*program, (cmdarg1, cmdarg2, ...)*).

os.execlp, os.execvp

Добавление «p» к execv или execl означает, что Python станет искать каталог, где находится программа, используя системный путь поиска (т. е. PATH).

os.execle, os.execve

Добавление «e» к execv или execl означает, что дополнительный *последний* аргумент является словарем, содержащим переменные оболочки, которые нужно передать программе.

os.execvpe, os.execlpe

Добавление как «p», так и «e» к базовым именам exec означает необходимость использования как пути поиска, так и словаря настроек окружения оболочки.

Поэтому когда сценарий в 3.3 вызывает os.execlp, отдельно передаваемые параметры задают командную строку для программы, которую нужно выполнить, а слово «python» отображается в исполняемый файл согласно установке пути поиска в системе ($PATH). Это соответствует выполнению в оболочке команды вида python child.py 1, но каждый раз с разными аргументами командной строки в конце.

Порожденная дочерняя программа

Так же как при вводе в оболочке, строка аргументов, передаваемая в os.execlp сценарием fork-exec в примере 3.3, запускает еще один файл программы Python, показанный в примере 3.4.

Пример 3.4. PP2E\System\Processes\child.py

```
import os, sys
print 'Hello from child', os.getpid(), sys.argv[1]
```

Вот как этот код выполняется в Linux. Он не сильно отличается от исходного *fork1.py*, но действительно выполняет новую программу в каждом ответвленном процессе. Более наблюдательные читатели заметят, что ID дочернего процесса одинаков в родительской программе и в запущенной программе *child.py* – os.execlp просто перегружает программу в том же самом процессе:

```
[mark@toy]$ python fork-exec.py
Child is 1094
Hello from child 1094 1

Child is 1095
Hello from child 1095 2

Child is 1096
```

```
Hello from child 1096 3
q
```

В Python есть и другие способы запуска программ, в том числе `os.system` и `os.popen`, с которыми мы познакомились в главе 2 «Системные инструменты» (для запуска командных строк оболочки), и вызов `os.spawnv`, с которым мы познакомимся далее в этой главе (для запуска независимых программ в Windows); эти темы, связанные с процессами, еще будут более глубоко рассмотрены в данной главе. Другие темы, относящиеся к процессам, будут рассматриваться и в последующих главах этой книги. Например, в главе 10 «Сценарии для сети» мы снова вернемся к ветвлению, чтобы разобраться с «зомби» – мертвыми процессами, затаившимися в системных таблицах после своего конца.

Потоки

Потоки представляют еще один способ начать действия, выполняемые одновременно. Иногда их называют «облегченными процессами», и работают они параллельно, как разветвленные процессы, но выполняются все в рамках одного и того же процесса. В приложениях, которые выигрывают от параллельной обработки, потоки дают программистам большие выгоды:

Производительность

Поскольку все потоки выполняются в одном процессе, для их запуска не требуется таких расходов, как при копировании процесса в целом. Издержки, связанные с копированием ответвляемых процессов и запуском потоков, могут быть различными в зависимости от платформы, но обычно считается, что потоки обходятся дешевле в смысле ущерба, наносимого производительности.

Простота

Программировать потоки оказывается значительно проще, особенно если на сцену выходят более сложные стороны процессов (например, завершение процессов, связь между процессами и процессы-«зомби», о которых рассказывается в главе 10).

Совместно используемая глобальная память

Кроме того, поскольку потоки выполняются в одном процессе, они используют общее для процесса пространство глобальной памяти. Благодаря этому потоки могут просто и естественно связываться друг с другом путем чтения и записи данных в глобальной памяти. Для программиста Python это означает, что глобальные (уровня модуля) переменные и компоненты интерпретатора совместно используются всеми потоками программы: если один поток присваивает значение глобальной переменной, ее новое значение видят все другие потоки. При обращении к совместно используемым глобальным объектам необходимо проявлять некоторую осторожность, но все равно это обычно проще, чем те средства связи, которые применяются для связи между ответвленными процессами и с которыми мы познакомимся ниже в этой главе (например, каналы, потоки, сигналы и т. д.).

Переносимость

Возможно, важнее всего то, что потоки лучше переносятся на другие платформы, чем процессы. На момент написания данной книги `os.fork` вообще не поддерживался в Windows, но потоки поддерживаются. В данное время, если в сценарии Python задачи должны выполняться параллельно и требуется переносимость, потоки окажутся, скорей всего, лучшим решением. Средства Python для работы с потоками автоматически учитывают специфические для каждой платформы различия в потоках и предоставляют единообразный интерфейс для всех операционных систем.

Использование потоков в Python осуществляется удивительно просто. В действительности при запуске программы уже начинает выполняться один поток, обычно называемый «главным потоком» процесса. Для запуска новых независимых потоков исполнения внутри процесса используется модуль Python `thread`, чтобы выполнить вызов функции в порожденном потоке, либо модуль Python `threading` для управления потоками с помощью объектов высокого уровня. Оба модуля предоставляют также средства для синхронизации доступа к совместно используемым объектам с помощью блокировок.

Модуль thread

Поскольку базовый модуль `thread` немного проще, чем более развитый модуль `threading`, о котором рассказывается далее в этом разделе, рассмотрим сначала его интерфейсы. Этот модуль предоставляет *переносимый* интерфейс к любой системе потоков, имеющейся на вашей платформе: его интерфейсы одинаково работают в Windows, Solaris, SGI и любой системе, где установлена реализация потоков POSIX «pthreads» (включая Linux). Сценарии Python, использующие модуль Python `thread`, работают на всех этих платформах без внесения каких-либо изменений в исходный код.

Начнем с экспериментирования со сценарием, демонстрирующим интерфейсы главного потока. Сценарий в примере 3.5 порождает потоки, пока не получит с консоли «q», и аналогичен по духу (будучи немного проще) сценарию в примере 3.1, но запускает параллельно потоки, а не ответвленные процессы.

Пример 3.5. PP2E\System\Threads\thread1.py

```
# порождать потоки до ввода с клавиатуры 'q'

import thread

def child(tid):
    print 'Hello from thread', tid

def parent():
    i = 0
    while 1:
        i = i+1
        thread.start_new(child, (i,))
        if raw_input() == 'q': break

parent()
```

В действительности в этом сценарии только две строчки имеют отношение к потокам: импорт модуля `thread` и вызов, создающий поток. Для создания потока просто выполняется функция `thread.start_new` независимо от того, на какой платформе осуществляется программирование.[1] Этот вызов принимает объект функции и набор аргументов и запускает новый поток, в котором будет выполняться переданная функция с переданными аргументами. Это очень похоже на встроенную функцию `apply` (и тут, и там принимается необязательный словарь аргументов с ключевыми словами), но в данном случае вызов функции начинает выполняться параллельно остальной части программы.

[1] По историческим причинам к этому вызову можно обращаться так же, как к `thread.start_new_thread`. Не исключено, что одно из двух имен одной и той же функции будет объявлено устаревшим в следующих версиях Python, но в примерах этого текста используются они оба. (В версии 2.2 `start_new` объявлена устаревшим синонимом `start_new_thread` – *Примеч. науч. ред.*)

Сам вызов thread.start_new сразу же выполняет возврат, не передавая какого-либо полезного значения, а порожденный им поток тихо завершается, когда происходит возврат из выполняемой функции (значение, возвращаемое выполняемой в потоке функцией, просто игнорируется). Кроме того, если выполняемая в потоке функция возбудит исключительную ситуацию, выводится трассировка стека и поток завершается, но остальная программа продолжает выполняться.

На практике, однако, использование потоков в сценарии Python почти тривиально. Прогоним эту программу, чтобы она запустила несколько новых потоков. На этот раз ее можно выполнять как в Linux, так и в Windows, потому что потоки переносятся лучше, чем ветвление процессов:

```
C:\...\PP2E\System\Threads>python thread1.py
Hello from thread 1

Hello from thread 2

Hello from thread 3

Hello from thread 4
q
```

Здесь каждое сообщение выводится новым потоком, который завершается почти сразу после своего начала. Чтобы действительно ощутить мощь параллельного выполнения потоков, нужно сделать в них что-либо более долго живущее. Порадую вас тем, что в Python играть с потоками столь же легко, как и приятно. Изменим программу fork-count из предыдущего раздела так, чтобы в ней использовались потоки. В сценарии примера 3.6 запускается 10 экземпляров counter, которые выполняются в параллельных потоках.

Пример 3.6. PP2E\System\Threads\thread-count.py

```
################################################################################
# основы потоков: запустить параллельно 10 экземпляров функции; используется
# time.sleep, чтобы главный поток не умер слишком рано – это убьет все другие потоки
# как в Windows, так и в Linux; stdout общий: выдача потоков может перемешаться
################################################################################
import thread, time

def counter(myId, count):                    # эта функция выполняется в потоках
    for i in range(count):
        #time.sleep(1)
        print '[%s] => %s' % (myId, i)

for i in range(10):                          # породить 10 потоков
    thread.start_new(counter, (i, 3))        # каждый поток проходит в цикле 3 раза

time.sleep(4)
print 'Main thread exiting.'                 # задержать завершение
```

Каждый параллельно выполняющийся экземпляр функции counter просто считает здесь от нуля до двух. При выполнении под Windows все 10 потоков выполняются одновременно, поэтому их выдача перемешана в стандартном выходном потоке:

```
C:\...\PP2E\System\Threads>python thread-count.py
... некоторые строки удалены ...
[5] => 0
[6] => 0
[7] => 0
[8] => 0
[9] => 0
```

```
[3] => 1
[4] => 1
[1] => 0
[5] => 1
[6] => 1
[7] => 1
[8] => 1
[9] => 1
[3] => 2
[4] => 2
[1] => 1
[5] => 2
[6] => 2
[7] => 2
[8] => 2
[9] => 2
[1] => 2
Main thread exiting.
```

В действительности выдача этих потоков перемешана в произвольном порядке, по крайней мере, в Windows. Она может даже иметь *различный* порядок при каждом запуске этого сценария. Поскольку все 10 потоков выполняются как независимые сущности, действительный порядок их перекрытия по времени зависит от почти случайного состояния системы в целом в момент их выполнения.

Если вы захотите сделать эту выдачу несколько более согласованной, раскомментируйте вызов (то есть удалите предшествующий `#`) `time.sleep(1)` в функции `counter` и снова запустите сценарий. Если сделать это, каждый из 10 потоков станет останавливаться на одну секунду перед тем, как вывести текущее значение счетчика. Из-за этой паузы все потоки в одинаковые моменты возвращают одинаковые числа; в действительности получается секундная задержка перед выводом каждой группы из 10 строк выдачи:

```
C:\...\PP2E\System\Threads>python thread-count.py
... некоторые строки удалены ...
[7] => 0
[6] => 0                                    остановка ...
[0] => 1
[1] => 1
[2] => 1
[3] => 1
[5] => 1
[7] => 1
[8] => 1
[9] => 1
[4] => 1
[6] => 1                                    остановка ...
[0] => 2
[1] => 2
[2] => 2
[3] => 2
[5] => 2
[9] => 2
[7] => 2
[6] => 2
[8] => 2
[4] => 2
Main thread exiting.
```

Однако даже при действии синхронизации с помощью паузы нельзя сказать, в каком порядке потоки выведут текущее значение счетчика. Он умышленно случаен: весь смысл запуска потоков в том, чтобы заставить работу выполняться независимо, параллельно.

Обратите внимание на то, что этот сценарий засыпает в конце на четыре секунды. Оказывается, по крайней мере в моих установках Windows и Linux, что главному потоку нельзя завершиться, пока выполняются порожденные им потоки: если он это сделает, немедленно завершаются порожденные потоки. Если бы не sleep, порожденные потоки погибали бы почти сразу. Может показаться, что так сделано специально, но это необходимо не на всех платформах, и программы обычно имеют такую структуру, чтобы главный поток естественным образом жил столько же, сколько потоки, которые им порождены. Например, интерфейс пользователя может начать выполнение загрузки FTP в потоке, но жизнь загрузки значительно короче, чем самого интерфейса пользователя. Далее в этом разделе мы покажем, как различными способами можно избежать этой паузы с помощью глобальных флагов, а также встретимся с утилитой «join» в другом модуле, которая позволяет явно ожидать завершения порожденных потоков.

Синхронизация доступа к глобальным объектам

Приятной особенностью потоков является наличие готового механизма связи между задачами: *совместно используемой глобальной памяти*. Например, поскольку все потоки выполняются в одном и том же процессе, то когда один поток Python изменяет глобальную переменную, это изменение видно всем другим потокам в процессе – главному и дочерним. Это дает простое средство передачи потоками программы информации друг другу – флаги завершения, результирующие объекты, индикаторы событий и т. д.

Недостатком такой схемы является то, что потоки должны следить за тем, чтобы не изменять глобальные объекты одновременно: если два потока одновременно изменяют объект, может случиться так, что одно из двух изменений окажется утраченным (или, что еще хуже, совместно используемый объект придет в полностью негодное состояние). Степень проблематичности этого зависит от приложения; иногда проблем вообще не возникает.

Но риску могут подвергаться и неочевидные вещи. Например, файлы и потоки данных (streams) совместно используются всеми потоками программы; если несколько потоков исполнения одновременно производят запись в один и тот же поток данных, в последнем могут появиться перемежающиеся искаженные данные. Продемонстрируем это: отредактируйте пример 3.6, закомментировав вызов sleep в `counter` и увеличив параметр `count` для каждого потока с 3 до 100, и в результате можно иногда увидеть в Windows такие странные результаты:

```
C:\...\PP2E\System\Threads\>python thread-count.py | more
... остальные удалены ...
[5] => 14
[7] => 14
[9] => 14
[3] => 15
[5] => 15
[7] => 15
[9] => 15
[3] => 16 [5] => 16 [7] => 16 [9] => 16


[3] => 17
```

```
[5] => 17
[7] => 17
[9] => 17
... остальные удалены ...
```

Поскольку все 10 потоков одновременно пытаются вести запись в `stdout`, время от времени на одну и ту же строчку попадает выдача из более чем одного потока. Такая особенность этого сценария необязательно приведет к тому, что спускаемый аппарат Mars Lander разобьется, но показывает, какого рода конфликты по времени могут происходить при параллельном выполнении программ. Чтобы работать надежно, программы, использующие потоки выполнения, должны управлять доступом к таким совместно используемым глобальным объектам, чтобы в один момент их использовал только один поток.[1]

К счастью, у модуля Python `thread` есть собственные простые в использовании средства для синхронизации доступа потоков к совместно используемым объектам. Эти средства основаны на понятии блокировки: чтобы изменить совместно используемый объект, потоки захватывают блокировку (*acquire lock*), производят требуемые изменения и снимают блокировку (*release lock*), которой после этого могут пользоваться другие потоки. Объекты блокировки размещаются в памяти, обрабатываются с помощью простых и переносимых вызовов в модуле `thread` и автоматически переносятся на механизмы блокировки потоков, существующие на соответствующей платформе.

Скажем, в примере 3.7 блокирующий объект, созданный `thread.allocate_lock`, захватывается каждым потоком перед утверждением `print`, с помощью которого осуществляется вывод в стандартный выходной поток, и освобождается после его выполнения.

Пример 3.7. PP2E\System\Threads\thread-count-mutex.py

```
#########################################################
# синхронизация доступа к stdout: поскольку это общий
# глобальный объект, выдача потоков может перемешаться
#########################################################
import thread, time

def counter(myId, count):
    for i in range(count):
        mutex.acquire()
        #time.sleep(1)
        print '[%s] => %s' % (myId, i)
        mutex.release()

mutex = thread.allocate_lock()
for i in range(10):
    thread.start_new_thread(counter, (i, 3))

time.sleep(6)
print 'Main thread exiting.'
```

Python гарантирует, что в каждый данный момент только один поток может захватить блокировку; все другие потоки, запрашивающие блокировку, останавливаются, пока освобождение блокировки вновь не сделает возможным ее захват. В результате добавления вызова блокировки в данный сценарий никакие два потока не смогут выполнять утверждение `print` в один и тот же момент времени: блокировка обеспечивает взаимоисключающий (*mutually exclusive*) доступ к потоку `stdout`. Поэтому выдача данного

[1] Более детальное объяснение этого явления можно найти далее во врезке «Глобальная блокировка интерпретатора и потоки».

сценария будет такой же, как у исходного *thread_count.py*, за исключением того, что текст на стандартном устройстве вывода не будет испорчен перекрывающимися `print`.

В данном случае раскомментирование вызова `time.sleep` в такой версии функции `counter` приведет к появлению каждой строчки выдачи с секундным интервалом. Поскольку поток засыпает, удерживая блокировку, все остальные потоки останавливается, пока владелец блокировки спит. Один поток захватывает блокировку, спит одну секунду и выводит строку; затем другой поток захватывает блокировку, спит и выводит строку, и т. д. Поскольку 10 потоков считают до 3, программа в целом заканчивается за 30 секунд (10×3), при этом каждую секунду выводится одна строка. Конечно, при этом предполагается, что главный поток спит не меньше этого времени. Чтобы узнать, как избавиться от этого предположения, нужно перейти к следующему разделу.

Глобальная блокировка интерпретатора и потоки

Строго говоря, в настоящее время Python использует механизм глобальной блокировки интерпретатора (*global interpreter lock*), обеспечивающий выполнение интерпретатором Python в каждый данный момент времени кода не более чем одного потока. Кроме того, чтобы дать возможность выполняться всем потокам, интерпретатор автоматически переключается между разными потоками через равные промежутки времени (путем освобождения и захвата блокировки после некоторого числа выполненных команд байт-кода), а также в начале длительных операций (например, файлового ввода/вывода).

Такая схема позволяет избежать проблем, могущих возникнуть при необходимости нескольким потокам одновременно обновить системные данные Python. К примеру, если бы двум потокам было разрешено одновременно изменить счетчик ссылок на объект, результаты оказались бы непредсказуемыми. Применение этой схемы может также повлечь тонкие неочевидные последствия. В частности, в примерах использования потоков, приведенных в данной главе, нарушения в `stdout` могут возникать потому, что вызов вывода текста в потоке исполнения является длительной операцией, которая влечет переключение потоков в интерпретаторе. В результате разрешается выполнение других потоков, которые делают запрос записи, когда в процессе выполнения еще находится предшествующая запись.

Кроме того, несмотря на то что глобальная блокировка интерпретатора не разрешает одновременного выполнения более чем одного потока Python, этого недостаточно для обеспечения безопасности потоков в целом и никак не решает проблемы синхронизации на более высоком уровне. Например, если одновременно несколько потоков пытаются *обновить* одну и ту же переменную, им обычно должен предоставляться исключительный доступ к объекту с помощью блокировок. В противном случае существует возможность того, что переключение потоков произойдет посреди байт-кода выражения, осуществляющего обновление. Рассмотрим такой код:

```
import thread, time
count = 0
def adder():
     global count
     count = count + 1          # одновременное обновление общей глобальной переменной
     count = count + 1          # поток вытесняется в середине обновления
for i in range(100):
     thread.start_new(adder, ())   # запустить 100 потоков обновления
```

```
time.sleep(5)
print count
```

В таком виде код будет неправильно работать в Windows из-за способа чередования его потоков (каждый раз будут получаться разные результаты, не 200), но будет работать, если вокруг операторов сложения разместить вызовы захвата/освобождения блокировки. Блокировки не обязательны для обращения ко всем совместно используемым объектам, особенно если один поток обновляет объект, а все остальные только наблюдают за изменениями. Однако на практике обычно следует использовать блокировки для синхронизации потоков в тех случаях, когда возможна их встреча при обновлении, а не полагаться на текущий способ реализации потоков.

Интересно, что приведенный выше код работает и в том случае, если сделать интервал переключения потоков достаточно большим, чтобы каждый из потоков мог выполниться прежде, чем будет переключен. Вызов `sys.setcheckinterval(N)` устанавливает частоту, с которой интерпретатор выполняет такие вещи, как переключение потоков и обработка сигналов. По умолчанию этот интервал равен 10, что задает число инструкций байт-кода, выполняемых между переключениями. В большинстве программ не требуется изменять эту частоту, но с ее помощью можно регулировать производительность работы потоков. Установка более высоких значений приводит к тому, что переключение происходит реже и потоки требуют меньших накладных расходов, но медленнее реагируют на события.

Если вы собираетесь использовать смешанный код Python и C, посмотрите также интерфейсы потоков, описываемые в стандартном руководстве по API Python/C. В многопоточных программах расширения C должны освобождать и снова захватывать глобальную блокировку интерпретатора вокруг длительных операций, чтобы позволить выполняться другим потокам Python.

Ожидание завершения порожденных потоков

Блокировки модуля потоков оказываются удивительно полезными. Они могут лечь в основу парадигм синхронизации более высокого уровня (например, семафоров) и использоваться как общие инструменты связи между потоками.[1] В частности, в примере 3.8 используется глобальный список блокировок для определения окончания работы всех дочерних потоков.

Пример 3.8. PP2E\System\Threads\thread-count-wait1.py

```
################################################################################
# использует мьютексы[2] для определения завершения потоков в родительском/главном
# потоке вместо time.sleep; блокирует stdout, чтобы печатать в 1 строке 1 раз;
################################################################################
```

[1] Однако их нельзя использовать для непосредственной синхронизации процессов. Поскольку процессы более независимы, им обычно требуются дольше живущие и внешние по отношению к программам механизмы блокировки. В главе 14 «Более крупные примеры сайтов, часть 2» мы познакомимся с библиотечным вызовом `fcntl.flock`, позволяющим сценариям блокировать и разблокировать файлы и являющимся поэтому идеальным средством блокировки между процессами.

[2] Мьютекс – специальный синхронизирующий объект в межпроцессном взаимодействии, подающий сигнал, когда он не захвачен каким-либо потоком. – *Примеч. ред.*

```
import thread

def counter(myId, count):
    for i in range(count):
        stdoutmutex.acquire()
        print '[%s] => %s' % (myId, i)
        stdoutmutex.release()
    exitmutexes[myId].acquire()    # сигнал главному потоку

stdoutmutex = thread.allocate_lock()
exitmutexes = []
for i in range(10):
    exitmutexes.append(thread.allocate_lock())
    thread.start_new(counter, (i, 100))

for mutex in exitmutexes:
    while not mutex.locked(): pass
print 'Main thread exiting.'
```

Для проверки состояния блокировки можно использовать ее метод `locked`. При работе главный поток создает по одной блокировке для каждого дочернего потока, помещая ее в глобальный список `exitmutexes` (помните, что функция потока использует глобальную область совместно с главным потоком). По завершении каждый поток захватывает свою блокировку в списке, а главный поток просто ждет, когда будут захвачены все блокировки. Это значительно более точный подход, чем наивно спать, пока выполняются дочерние потоки, в надежде, проснувшись, обнаружить, что все они завершены.

Но погодите, все еще проще: поскольку потоки в любом случае совместно используют глобальную память, того же результата можно добиться с помощью простого глобального списка *целых чисел*, а не блокировок. В примере 3.9 пространство имен модуля (область видимости) совместно используется кодом верхнего уровня и функцией, выполняемой в потоке, как и ранее: имя `exitmutexes` указывает на один и тот же объект списка в главном потоке и всех порождаемых потоках. По этой причине изменения, производимые в потоке, видны в главном потоке без использования лишних блокировок.

Пример 3.9. PP2E\System\Threads\thread-count-wait2.py

```
################################################################################
# использует обычные глобальные данные с совместным доступом (не мьютексы)
# для определения главным потоком момента завершения дочерних потоков;
################################################################################
import thread
stdoutmutex = thread.allocate_lock()
exitmutexes = [0] * 10
def counter(myId, count):
    for i in range(count):
        stdoutmutex.acquire()
        print '[%s] => %s' % (myId, i)
        stdoutmutex.release()
    exitmutexes[myId] = 1  # сигнал главному потоку
for i in range(10):
    thread.start_new(counter, (i, 100))
while 0 in exitmutexes: pass
print 'Main thread exiting.'
```

Главные потоки в конце обоих сценариев попадают в напряженный цикл ожидания, который может заметно снизить производительность в критических по времени приложениях. В таких случаях нужно просто добавить в цикл ожидания вызов `time.sleep`,

чтобы между проверками образовалась пауза и ЦП освободился для других задач. Потоки тоже должны быть добропорядочными гражданами.

Оба последних сценария с потоками-счетчиками производят примерно такую же выдачу, как первоначальный *thread_count.py*, хотя и без повреждения `stdout` и с различным случайным порядком выходных строк. Основное различие в том, что главный поток завершает работу сразу после (и не раньше!) порожденных дочерних потоков:

```
C:\...\PP2E\System\Threads>python thread-count-wait2.py
... остальные удалены ...
[2] => 98
[6] => 97
[0] => 99
[7] => 97
[3] => 98
[8] => 97
[9] => 97
[1] => 99
[4] => 98
[5] => 98
[2] => 99
[6] => 98
[7] => 98
[3] => 99
[8] => 98
[9] => 98
[4] => 99
[5] => 99
[6] => 99
[7] => 99
[8] => 99
[9] => 99
Main thread exiting.
```

Конечно, потоки способны на значительно большее, чем ведение счета. В одной из следующих глав такие совместно используемые глобальные данные будут применены с более практическими целями в качестве сигналов, посылаемых для вывода интерфейсом пользователя Tkinter GUI при завершении дочерних процессов, передающих данные по сети (смотрите «Клиент электронной почты PyMailGui» в главе 11 «Сценарии, выполняемые на стороне клиента»).

Модуль threading

В стандартную библиотеку Python входят два модуля для работы с потоками: `thread` – основной интерфейс нижнего уровня, который показывался до сих пор, и `threading` – интерфейс более высокого уровня, основанный на объектах. Внутри себя модуль `threading` использует модуль `thread` для реализации объектов, представляющих потоки и общие средства синхронизации. Он в какой-то мере основан на подмножестве модели потоков Java, но есть различия, которые заметят только программисты Java.[1] В примере 3.10 наш сценарий с потоками-счетчиками видоизменен еще один последний раз, чтобы показать интерфейсы этого нового модуля.

[1] На случай, если вы таковым и являетесь: переменные Python lock и condition являются отдельными объектами, а не чем-то присущим всем объектам, а класс Python `Thread` обладает не всеми характеристиками, которые есть у него в Java. Дополнительные детали можно найти в руководстве по библиотеке Python.

Пример 3.10. PP2E\System\Threads\thread-classes.py

```
##############################################################################
# использует метод join java-подобного объекта модуля threading (вместо мьютексов
# и глобальных переменных), чтобы родительский/главный поток мог узнать о завершении
# потоков; подробнее о threading в руководстве по библиотеке;
##############################################################################

import threading

class mythread(threading.Thread):               # подкласс объекта Thread
    def __init__(self, myId, count):
        self.myId  = myId
        self.count = count
        threading.Thread.__init__(self)
    def run(self):                              # run обеспечивает логику потока
        for i in range(self.count):             # синхронизировать доступ к stdout
            stdoutmutex.acquire()
            print '[%s] => %s' % (self.myId, i)
            stdoutmutex.release()

stdoutmutex = threading.Lock()                  # то же, что thread.allocate_lock()
threads = []
for i in range(10):
    thread = mythread(i, 100)                   # создать/запустить 10 потоков
    thread.start()                              # запустить метод run в потоке
    threads.append(thread)

for thread in threads:
    thread.join()                               # ждать завершения потока
print 'Main thread exiting.'
```

Выдача этого сценария такая же, как у его предшественников (снова случайно распределенная). Использование `threading` состоит в основном из определения классов. Потоки в этом модуле реализуются с помощью объекта `Thread` – класса Python, который настраивается в каждом приложении путем задания метода `run`, определяющего действия, выполняемые потоком. Например, в данном сценарии создается подкласс `Thread` с именем `mythread`; метод `run` класса `mythread` и будет выполняться в рамках `Thread`, если создать `mythread` и вызвать его метод `start`.

Иными словами, этот сценарий просто обеспечивает методы, предполагаемые структурой `Thread`. Этот путь, требующий большего объема кодирования, дает то преимущество, что открывается «бесплатный» доступ к ряду дополнительных средств работы с потоками, предоставляемых данной структурой. К примеру, используемый в конце сценария метод `Thread.join` ожидает завершения (по умолчанию) потока; с его помощью можно не дать главному потоку завершиться слишком рано, отказавшись от вызова `time.sleep`, глобальных блокировок и переменных, использовавшихся в предыдущих примерах с потоками.

Сценарий примера использует также `threading.Lock` для синхронизации доступа к выходному потоку (хотя в настоящей реализации это просто синоним для `thread.allocate_lock`). Помимо `Thread` и `Lock` модуль `threading` содержит объекты высокого уровня для синхронизации доступа к совместно используемым элементам (например, `Semaphore`, `Condition`, `Event`) и другие вещи; подробности можно найти в руководстве по библиотеке. Другие примеры потоков и ветвления в целом есть в следующем разделе и в примерах части III «Сценарии для Интернета».

Завершение программ

Как мы видели, в отличие от C, в Python нет функции «main» – при запуске программы просто выполняется с начала и до конца весь код файла верхнего уровня (то есть файла, указанного в командной строке, запущенного из проводника и т. д.). Обычно сценарии завершаются, когда Python проходит конец файла, но завершить программу можно и явно с помощью встроенной функции `sys.exit`:

```
>>> sys.exit()           # иначе завершение по концу сценария
```

Интересно, что этот вызов в действительности только возбуждает встроенную исключительную ситуацию `SystemExit`. Благодаря этому его можно обычным образом перехватывать, чтобы выполнить завершающие действия; если он не перехвачен, интерпретатор завершает работу обычным способом. Например:

```
C:\...\PP2E\System>python
>>> import sys
>>> try:
...     sys.exit()              # см. также: os._exit, Tk().quit()
... except SystemExit:
...     print 'ignoring exit'
...
ignoring exit
>>>
```

В действительности явное возбуждение встроенной исключительной ситуации `SystemExit` с помощью утверждения Python `raise` эквивалентно вызову `sys.exit`. В практических сценариях блок `try` должен перехватывать исключительную ситуацию завершения работы, возбужденную в любом месте программы; сценарий примера 3.11 завершается из выполняющейся функции.

Пример 3.11. PP2E\System\Exits\testexit_sys.py

```
def later():
    import sys
    print 'Bye sys world'
    sys.exit(42)
    print 'Never reached'

if __name__ == '__main__': later()
```

Выполнение этой программы как сценария заставляет ее завершиться прежде, чем интерпретатор проскочит конец файла. Но поскольку `sys.exit` возбуждает исключительную ситуацию Python, при импорте этой функции можно перехватывать вызываемую в ней исключительную ситуацию завершения и отменять его либо задавать блок `finally`, который должен быть выполнен при завершении программы:

```
C:\...\PP2E\System\Exits>python testexit_sys.py
Bye sys world

C:\...\PP2E\System\Exits>python
>>> from testexit_sys import later
>>> try:
...     later()
... except SystemExit:
...     print 'Ignored...'
...
Bye sys world
Ignored...
```

```
>>> try:
...     later()
... finally:
...     print 'Cleanup'
...
Bye sys world
Cleanup

C:\...\PP2E\System\Exits>
```

Завершение работы с помощью модуля os

Можно выйти из Python и другими способами. Например, в ответвленном дочернем процессе в Unix обычно вызывается `os._exit`, а не `sys.exit`, потоки могут завершаться с помощью вызова `thread.exit`, а приложения Tkinter GUI часто завершаются с помощью такого вызова, как `Tk().quit()`. С модулем Tkinter мы познакомимся далее в этой книге, но `os` и `thread` заслуживают того, чтобы мы рассмотрели их сейчас. При вызове `os._exit` вызывающий процесс завершается сразу, не возбуждая исключительной ситуации, которую можно перехватывать и игнорировать, как показано в примере 3.12.

Пример 3.12. PP2E\System\Exits\testexit_os.py

```
def outahere():
    import os
    print 'Bye os world'
    os._exit(99)
    print 'Never reached'

if __name__ == '__main__': outahere()
```

В отличие от `sys.exit`, на `os._exit` не действует перехват ни `try/except`, ни `try/finally`.

```
C:\...\PP2E\System\Exits>python testexit_os.py
Bye os world

C:\...\PP2E\System\Exits>python
>>> from testexit_os import outahere
>>> try:
...     outahere()
... except:
...     print 'Ignored'
...
Bye os world

C:\...\PP2E\System\Exits>python
>>> from testexit_os import outahere
>>> try:
...     outahere()
... finally:
...     print 'Cleanup'
...
Bye os world
```

Коды состояния завершения

Оба вызова завершения `sys` и `os`, с которыми мы только что познакомились, принимают аргумент, обозначающий код состояния завершения процесса (в вызове `sys` он необязателен, но в `os` необходим). После завершения этот код может запрашиваться оболочкой или программой, запустившей сценарий как дочерний процесс. В Linux за-

прашивается значение переменной оболочки «status», чтобы получить состояние завершения последней программы; обычно ненулевое состояние указывает на то, что возникли какие-то проблемы:

```
[mark@toy]$ python testexit_sys.py
Bye sys world
[mark@toy]$ echo $status
42
[mark@toy]$ python testexit_os.py
Bye os world
[mark@toy]$ echo $status
99
```

В цепочке программ командной строки попутная проверка состояний завершения может использоваться как простая форма связи между программами. Можно также получить состояние завершения программы, выполненной другим сценарием. При запуске команд оболочки оно предоставляется как значение, возвращаемое вызовом `os.system`, и значение, возвращаемое методом `close` объекта `os.popen`; при ветвлении программ состояние завершения доступно через вызовы `os.wait` и `os.waitpid` в родительском процессе. Сначала рассмотрим случай команд оболочки:

```
[mark@toy]$ python
>>> import os
>>> pipe = os.popen('python testexit_sys.py')
>>> pipe.read()
'Bye sys world\012'
>>> stat = pipe.close()              # возвращает код завершения
>>> stat
10752
>>> hex(stat)
'0x2a00'
>>> stat >> 8
42

>>> pipe = os.popen('python testexit_os.py')
>>> stat = pipe.close()
>>> stat, stat >> 8
(25344, 99)
```

При использовании `os.popen` состояние завершения в действительности представляется как специальные битовые позиции возвращаемого значения – по причинам, в которые мы сейчас не станем вникать. Оно действительно находится там, но чтобы его увидеть, нужно сдвинуть результат вправо на восемь разрядов. Команды, выполняемые через `os.system`, возвращают свое состояние через библиотечный вызов Python:

```
>>> import os
>>> for prog in ('testexit_sys.py', 'testexit_os.py'):
...     stat = os.system('python ' + prog)
...     print prog, stat, stat >> 8
...
Bye sys world
testexit_sys.py 10752 42
Bye os world
testexit_os.py 25344 99
```

К сожалению, когда я писал эти строки, интерфейсы popen и system для получения состояния завершения работали в Windows ненадежно. Более того, там вообще не поддерживается fork, а popen в Python 1.5.2 и более ранних дает отказ в приложениях, создающих окна (хотя работает в коде, выполняемом из командных строк консоли DOS, и в целом лучше работает в версии 2.0). В Windows:

```
>>> import os
>>> stat = os.system('python testexit_sys.py')
Bye sys world
>>> print stat
0
>>> pipe = os.popen('python testexit_sys.py')
>>> print pipe.read(),
Bye sys world
>>> print pipe.close()
None
>>> os.fork
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: fork
```

В настоящее время для решения таких задач может потребоваться применение специфических для Windows инструментов (например, os.spawnv и выполнение команды DOS start с помощью os.system; смотрите далее в этой главе). Однако следите за изменениями в этом направлении: в Python 2.0 исправлены проблемы Windows с popen, а ActiveState, компания, создавшая вызов fork для Perl в Windows, начала сосредоточиваться на разработке инструментария для Python.

Чтобы научиться получать статус завершения ответвленных процессов, напишем простую программу для ветвления: сценарий в примере 3.13 ответвляет дочерние процессы и выводит состояния завершения дочернего процесса, возвращаемые вызовом os.wait в родительском процессе, пока с консоли не будет введена «q».

Пример 3.13. PP2E\System\Exits\testexit_fork.py

```
########################################################################################
# ответвление дочерних процессов для получения состояния завершения с
# помощью os.wait; в Python 1.5.2 ветвление работает в Linux, но не в Windows
# замечание: порожденные процессы совместно используют глобальные переменные,
# но каждый ответвленный процесс хранит собственный их экземпляр - exitstat
# здесь всегда одинаков, но будет различаться, если вместо этого запускать потоки;
########################################################################################
import os
exitstat = 0

def child():                            # здесь можно сделать os.exit из сценария
    global exitstat                     # изменить глобальную переменную этого процесса
    exitstat = exitstat + 1             # состояние завершения в родительский wait
    print 'Hello from child', os.getpid(), exitstat
    os._exit(exitstat)
    print 'never reached'

def parent():
```

```
    while 1:
        newpid = os.fork()                  # запустить новый экземпляр процесса
        if newpid == 0:                     # если копия, выполняется дочерний процесс
            child()                         # цикл до ввода 'q' с консоли
        else:
            pid, status = os.wait()
            print 'Parent got', pid, status, (status >> 8)
            if raw_input() == 'q': break

parent()
```

Запуск этой программы под Linux (помните, `fork` не работал под Windows, когда писалось второе издание этой книги) приводит к следующим результатам:

```
[mark@toy]$ python testexit_fork.py
Hello from child 723 1
Parent got 723 256 1

Hello from child 724 1
Parent got 724 256 1

Hello from child 725 1
Parent got 725 256 1
q
```

Если внимательно изучить эту выдачу, можно заметить, что состояние завершения (последнее выводимое число) всегда одинаково – 1. Поскольку ответвленные процессы начинают жизнь как *копии* создавших их процессов, у них есть и копия глобальной памяти. Поэтому каждый ответвившийся дочерний процесс получает и изменяет собственную глобальную переменную `exitstat`, не трогая экземпляров этой переменной в других процессах.

Завершение потоков

В отличие от процессов, потоки выполняются параллельно внутри *одного и того же* процесса и совместно используют глобальную память. Все потоки в примере 3.14 изменяют одну и ту же глобальную переменную `exitstat`.

Пример 3.14. PP2E\System\Exits\testexit_thread.py

```
########################################################################
# породить потоки и следить за изменениями глобальной памяти;
# обычно потоки завершаются при возврате из выполняемой в них функции,
# но можно завершить поток, вызвав thread.exit(); thread.exit - то же самое,
# что sys.exit, и возбуждает SystemExit; потоки связываются
# через глобальные переменные, которые можно блокировать;
########################################################################
import thread
exitstat = 0
def child():
    global exitstat                         # обработка глобальных имен,
    exitstat = exitstat + 1                 # совместно используемых всеми потоками
    threadid = thread.get_ident()
    print 'Hello from child', threadid, exitstat
    thread.exit()
    print 'never reached'

def parent():
    while 1:
        thread.start_new_thread(child, ())
```

```
        if raw_input() == 'q': break

parent()
```

Вот как этот сценарий действует в Linux: глобальная переменная `exitstat` изменяется каждым потоком, потому что потоки совместно используют глобальную память внутри процесса. В действительности потоки вообще часто общаются друг с другом таким способом – вместо кодов состояния завершения потоки присваивают глобальные переменные уровня модуля сигнальным условиям (и при необходимости с помощью блокировок модуля `thread` синхронизируют доступ к совместно используемым глобальным переменным):

```
[mark@toy]$ /usr/bin/python testexit_thread.py
Hello from child 1026 1

Hello from child 2050 2

Hello from child 3074 3
q
```

В отличие от ветвления, потоки можно выполнять под Windows – эта программа выполняется там так же, но отличаются идентификаторы потоков: они произвольны, но уникальны для всех активных потоков, и потому могут использоваться как ключи словаря для хранения данных, относящихся к каждому потоку:

```
C:\...\PP2E\System\Exits>python testexit_thread.py
Hello from child -587879 1

Hello from child -587879 2

Hello from child -587879 3
q
```

Если говорить о завершении, то для процесса оно обычно происходит бесшумно, а возвращаемое функцией значение игнорируется. Можно использовать функцию `thread.exit` для явного завершения вызвавшего ее процесса. Этот вызов работает почти в точности так же, как `sys.exit` (но не принимает аргумента состояния завершения), и действует путем возбуждения в вызвавшем потоке исключительной ситуации `SystemExit`. Поэтому поток можно также досрочно завершить, вызвав `sys.exit` или непосредственно возбудив `SystemExit`. Следите, однако, за тем, чтобы не вызывать `os._exit` внутри функции потока – на моей системе Linux в результате подвешивается весь процесс, а в Windows убиваются все потоки процесса!

При правильном использовании состояние завершения может участвовать в обнаружении ошибок и в простых коммуникационных протоколах в системах, образуемых сценариями командных строк. Но при этом я должен подчеркнуть, что большинство сценариев для завершения использует просто выход за конец исходного файла, а из большинства функций потоков просто происходит возврат; явный вызов завершения обычно применяется только для исключительных ситуаций.

Межпроцессное взаимодействие

Ранее мы видели, что при порождении сценариями *потоков* – задач, выполняемых параллельно внутри программы – последние могут естественным образом поддерживать связь друг с другом путем изменения и чтения совместно используемой глобальной памяти. Мы видели также, что следует подумать об использовании блокировок для синхронизации доступа к совместно используемым объектам, которые могут быть модифицированы одновременно, но такая модель связи очень проста.

Все становится сложнее, когда сценарии запускают процессы и программы. Если ограничиться теми видами связи, которые могут осуществляться между программами, то здесь существует много вариантов, большинство из которых уже было продемонстрировано в этой и предыдущих главах. Например, в качестве средств коммуникации между программами могут рассматриваться:

- аргументы командной строки
- переадресация стандартных потоков
- каналы, создаваемые вызовами `os.popen`
- коды состояния завершения программ
- переменные окружения оболочки
- даже простые файлы

Например, задание опций в командной строке и запись во входные потоки позволяет передавать параметры выполнения программы; чтение выходных потоков и кодов завершения дает возможность получать результаты. Поскольку установки переменных оболочки наследуются порождаемыми программами, они дают еще один способ задания контекста. Каналы, создаваемые при помощи вызова `os.popen`, и простые файлы позволяют осуществлять еще более динамическую связь: программы могут обмениваться данными в произвольные моменты времени, не только во время запуска или завершения.

Помимо указанной группы в библиотеке Python есть и другие средства для осуществления IPC – Inter-Process Communication (межпроцессного взаимодействия). Некоторые из них имеют особенности, связанные с переносимостью, и все они различаются по сложности. Например, в главе 10 мы познакомимся с модулем Python `socket`, который позволяет обмениваться данными как программам, выполняющимся на одном компьютере, так и программам, находящимся на разных машинах сети.

В данном разделе мы познакомимся с каналами (*pipes*) – анонимными и именованными, а также сигналами (*signals*) – средствами генерации событий в других программах. Программисты Python могут пользоваться и другими средствами IPC (например, совместно используемой памятью; смотрите модуль `mmap`), но о которых здесь не рассказывается из-за недостатка места; если вас интересует что-либо более специфическое, ищите в руководствах Python и на веб-сайте подробности использования других схем IPC.

Каналы

Доступ к каналам, еще одному средству связи между программами, осуществляется в Python с помощью встроенного вызова `os.pipe`. Каналы направлены в одну сторону, и их действие напоминает буфер совместно используемой памяти, интерфейс которого с обеих сторон похож на простой файл. Типичное использование канала состоит в том, что одна программа пишет данные с одного конца канала, а вторая читает их с другого конца. Каждая из программ видит только свой конец канала и обрабатывает его с помощью обычных функций Python для работы с файлами.

Однако каналы по большей части принадлежат операционной системе. Например, вызовы для чтения из канала обычно блокируют вызвавшего, пока данные не станут доступны (т. е. будут отправлены программой на другом конце), вместо того чтобы возвратить признак конца файла. Такие особенности позволяют также использовать каналы для синхронизации выполнения независимых программ.

Основы анонимных каналов

Каналы бывают двух видов – анонимные (*anonymous*) и именованные (*named*). Именованные каналы (иногда называемые «fifo») представляются в компьютере в виде файла. Анонимные же каналы существуют только внутри процессов и обычно используются в месте с *ветвлением* процессов как средство связи родительского и порожденного дочернего процессов в приложении: родитель и потомок общаются через совместно используемые описатели файлов каналов. Поскольку именованные каналы являются в действительности внешними файлами, общающиеся друг с другом процессы могут быть никак не связаны друг с другом (в действительности они могут быть независимо запущенными программами).

Начнем с более традиционных каналов – анонимных. В качестве иллюстрации сценарий примера 3.15 использует вызов `os.fork`, чтобы, как обычно, создать копию вызывающего процесса (с ветвлением мы познакомились выше в этой главе). После ветвления исходный родительский процесс и его дочерняя копия разговаривают между собой с двух концов канала, созданного `os.pipe` перед ветвлением. Вызов `os.pipe` возвращает набор из двух дескрипторов файлов – идентификаторов файлов низкого уровня, с которыми мы уже знакомы, представляющими входной и выходной концы канала. Так как ответвленный дочерний процесс получает копию дескрипторов файлов своего родителя, то при записи в дескриптор вывода канала в дочернем процессе данные посылаются обратно родителю по каналу, созданному до порождения дочернего процесса.

Пример 3.15. PP2E\System\Processes\pipe1.py

```
import os, time
def child(pipeout):
    zzz = 0
    while 1:
        time.sleep(zzz)                          # родитель ждет
        os.write(pipeout, 'Spam %03d' % zzz)     # отправить родителю
        zzz = (zzz+1) % 5                        # переход к 0 после 4
def parent():
    pipein, pipeout = os.pipe()                  # создать канал с 2 концами
    if os.fork() == 0:                           # копировать процесс
        child(pipeout)                           # в копии запустить child
    else:                                        # в родителе слушать канал
        while 1:
            line = os.read(pipein, 32)           # останавливается до отправки данных
            print 'Parent %d got "%s" at %s' % (os.getpid(), line, time.time())
parent()
```

Если запустить эту программу под Linux (`pipe` сегодня есть под Windows, а `fork` – нет), то родительский процесс при каждом вызове `os.read` будет ждать, пока дочерний процесс отправит данные в канал. Здесь дочерний и родительский процессы действуют почти как клиент и сервер: родитель запускает дочерний процесс и ждет от него установки связи.[1] Дочерний процесс дразнит родителя тем, что заставляет его ожидать между сообщениями на одну секунду дольше с помощью вызова `time.sleep`, пока за-

[1] Понятия «клиент» и «сервер» будут разъяснены в главе 10. Там связь будет организовываться с помощью *сокетов* (которые можно грубо назвать двунаправленными каналами для сетей), но модель обмена информацией в целом остается схожей. Именованные каналы (fifo), описываемые ниже, лучше соответствуют модели клиент/сервер, поскольку доступ к ним может осуществляться произвольными несвязанными процессами (ветвления при этом не требуется). Но как мы увидим, модели сокетов повсеместно используются большинством протоколов сценариев для Интернета.

держка не достигнет четырех секунд. Когда счетчик задержки zzz становится равным 005, он сбрасывается обратно в 000 и начинает работать сначала:

```
[mark@toy]$ python pipe1.py
Parent 1292 got "Spam 000" at 968370008.322
Parent 1292 got "Spam 001" at 968370009.319
Parent 1292 got "Spam 002" at 968370011.319
Parent 1292 got "Spam 003" at 968370014.319
Parent 1292 got "Spam 004Spam 000" at 968370018.319
Parent 1292 got "Spam 001" at 968370019.319
Parent 1292 got "Spam 002" at 968370021.319
Parent 1292 got "Spam 003" at 968370024.319
Parent 1292 got "Spam 004Spam 000" at 968370028.319
Parent 1292 got "Spam 001" at 968370029.319
Parent 1292 got "Spam 002" at 968370031.319
Parent 1292 got "Spam 003" at 968370034.319
```

Если внимательно посмотреть, то видно, что когда счетчик задержки в дочернем процессе достигает 004, это приводит к считыванию родителем *сразу* двух сообщений из канала: дочерний процесс записал два различных сообщения, но они были достаточно близки по времени и получены родителем как один блок. В действительности родитель каждый раз слепо запрашивает чтение не более 32 байтов, но получает тот текст, который есть в канале (если он там есть вообще). Чтобы лучше отделять одно сообщение от другого, можно определить для канала символ-разделитель. Это легко сделать с помощью конца строки, так как можно заключить дескриптор канала в файловый объект с помощью `os.fdopen` и использовать метод объекта файла `readline` для поиска в канале очередного разделителя `\n`. Такая схема реализована в примере 3.16.

Пример 3.16. PP2E\System\Processes\pipe2.py

```
# то же, что pipe1.py, но ввод из канала заключен в файловый объект stdio
# для возможности построчного чтения и закрытия неиспользуемых
# дескрипторов файлов в обоих процессах

import os, time

def child(pipeout):
    zzz = 0
    while 1:
        time.sleep(zzz)                          # родитель ждет
        os.write(pipeout, 'Spam %03d\n' % zzz)   # отправить родителю
        zzz = (zzz+1) % 5                        # переход к 0 при 5

def parent():
    pipein, pipeout = os.pipe()                  # создать канал с 2 концами
    if os.fork() == 0:                           # в дочернем процессе писать в канал
        os.close(pipein)                         # закрыть входной конец
        child(pipeout)
    else:                                        # родитель ждет данных в канале
        os.close(pipeout)                        # закрыть выходной конец
        pipein = os.fdopen(pipein)               # создать объект stdio для ввода
        while 1:
            line = pipein.readline()[:-1]        # останавливается до отправки данных
            print 'Parent %d got "%s" at %s' % (os.getpid(), line, time.time())

parent()
```

Этот вариант лучше и потому, что неиспользуемый конец канала в каждом процессе *закрывается* (например, после ветвления родительский процесс закрывает свой эк-

земпляр выходного конца канала, в который пишет дочерний процесс); обычно программы должны закрывать неиспользуемые концы каналов. При выполнении этой новой версии родителю при каждом чтении из канала возвращается одно сообщение дочернего процесса, потому что при записи все они разделяются маркерами:

```
[mark@toy]$ python pipe2.py
Parent 1296 got "Spam 000" at 968370066.162
Parent 1296 got "Spam 001" at 968370067.159
Parent 1296 got "Spam 002" at 968370069.159
Parent 1296 got "Spam 003" at 968370072.159
Parent 1296 got "Spam 004" at 968370076.159
Parent 1296 got "Spam 000" at 968370076.161
Parent 1296 got "Spam 001" at 968370077.159
Parent 1296 got "Spam 002" at 968370079.159
Parent 1296 got "Spam 003" at 968370082.159
Parent 1296 got "Spam 004" at 968370086.159
Parent 1296 got "Spam 000" at 968370086.161
Parent 1296 got "Spam 001" at 968370087.159
Parent 1296 got "Spam 002" at 968370089.159
```

Двунаправленный IPC с помощью каналов

Обычно каналы позволяют данным перемещаться только в *одном направлении* – один конец является входом, другой выходом. Что делать, если требуется общение между программами в обоих направлениях? Например, одна программа может посылать другой запрос на информацию и ждать получения этой информации. Один канал обычно не может справиться с такими двунаправленными переговорами, но два могут: один канал используется для передачи программе запроса, а второй – для пересылки ответа затребовавшей его программе.[1]

Модуль в примере 3.17 демонстрирует один из способов применения этой идеи для связи входных и выходных потоков двух программ. В нем функция `spawn` ответвляет новую дочернюю программу и соединяет входной и выходной потоки родительской программы с выходным и входным потоками дочерней. Это означает, что:

- при чтении родителем стандартного ввода происходит чтение текста, отправленного дочерней программой на свой стандартный вывод;
- при записи родителем на стандартный вывод происходит отправка данных на стандартный вывод дочерней программы.

В итоге две независимые программы обмениваются между собой данными через свои стандартные входные и выходные потоки.

Пример 3.17. PP2E\System\Processes\pipes.py

```
########################################################################################
# породить дочерний процесс/программу, соединить свой stdin/stdout со stdout/stdin
# дочернего процесса – чтение и запись родителя отображаются в выходной и входной потоки
```

[1] Это действительно имеет практические применения. Например, однажды я добавил графический интерфейс в отладчик командной строки для С-подобного языка программирования, соединив два процесса через каналы. GUI выполнялся как отдельный процесс, создававший команды и отправлявший их в существующий канал входного потока отладчика, производя анализ результатов, появлявшихся в канале выходного потока отладчика. В результате GUI выполнял роль программиста, вводящего команды с клавиатуры. Порождая программы командной строки, к потокам которых прикреплены каналы, можно создавать новые интерфейсы к старым программам.

```
# порожденной программы; похоже на popen2.popen2 плюс переадресация родительских потоков;
#########################################################################################
import os, sys
def spawn(prog, *args):                 # передача имени программы, аргументов командной строки
    stdinFd  = sys.stdin.fileno()             # получить описатели потоков
    stdoutFd = sys.stdout.fileno()            # обычно stdin=0, stdout=1

    parentStdin, childStdout  = os.pipe()     # создать два канала ipc
    childStdin,  parentStdout = os.pipe()     # pipe возвращает inputfd, outoutfd
    pid = os.fork()                           # создать копию процесса
    if pid:
        os.close(childStdout)                 # в родительском процессе после ветвления:
        os.close(childStdin)                  # закрыть дочерние концы в родителе
        os.dup2(parentStdin,  stdinFd)        # копия sys.stdin = pipe1[0]
        os.dup2(parentStdout, stdoutFd)       # копия sys.stdout = pipe2[1]
    else:
        os.close(parentStdin)                 # в дочернем процессе после ветвления:
        os.close(parentStdout)                # закрыть родительские концы в дочернем
        os.dup2(childStdin,  stdinFd)         # копия sys.stdin = pipe2[0]
        os.dup2(childStdout, stdoutFd)        # копия sys.stdout = pipe1[1]
        args = (prog,) + args
        os.execvp(prog, args)                 # новая программа в этом процессе
        assert 0, 'execvp failed!'            # вызов os.exec не возвращается сюда

if __name__ == '__main__':
    mypid = os.getpid()
    spawn('python', 'pipes-testchild.py', 'spam')    # ветвить дочернюю программу

    print 'Hello 1 from parent', mypid                # на stdin дочерней
    sys.stdout.flush()                                # сбросить буфер stdio
    reply = raw_input()                               # из stdout дочернего процесса
    sys.stderr.write('Parent got: "%s"\n' % reply)    # stderr не привязан к каналу!

    print 'Hello 2 from parent', mypid
    sys.stdout.flush()
    reply = sys.stdin.readline()
    sys.stderr.write('Parent got: "%s"\n' % reply[:-1])
```

Функция `spawn` в этом модуле не работает под Windows: помните, что `fork` там пока нет. В действительности вызовы из этого модуля прямо отображаются в системные вызовы Unix (и тех, кто не пишет для Unix, могут поначалу ужаснуть). С некоторыми из них мы уже встречались (например, `os.fork`), но значительная часть этого кода основывается на понятиях Unix, разобраться с которыми в данной книге должным образом нам не позволит время. В упрощенном виде краткий обзор системных вызовов, имеющихся в этом коде, выглядит так:

- **`os.fork` копирует, как обычно, вызывающий процесс и возвращает ID дочернего процесса в родительский процесс.**
- **`os.execvp` загружает в вызывающий процесс новую программу; это то же, что использовавшийся выше `os.execlp`, но принимает набор или список аргументов командной строки (получаемых в виде `*args` в заголовке функции).**
- **`os.pipe` возвращает набор описателей файлов, представляющих входной и выходной конец канала, как и в приведенных ранее примерах.**
- **`os.close(fd)` закрывает файл с описателем `fd`.**
- **`os.dup2(fd1,fd2)` копирует всю системную информацию, связанную с файлом, заданным описателем `fd1`, в файл, заданный `fd2`.**

В отношении соединения стандартных потоков вся кухня творится в os.dup2. Например, вызов os.dup2(parentStdin,stdinFd) по существу присваивает файл stdin родительского процесса входному концу одного из создаваемых каналов; всякое чтение stdin с этого момента происходит из канала. После соединения другого конца этого канала с копией файла потока stdout дочернего процесса посредством os.dup2(childStdout,stdoutFd) текст, выводимый дочерним процессом на его sdtdout, оказывается отправленным через канал в поток stdin его родителя.

Для проверки этой утилиты в конце файла помещен код самотестирования, который порождает в дочернем процессе программу, приведенную в примере 3.18, и производит чтение и запись в стандартные потоки для связи с ней через два канала.

Пример 3.18. PP2E\System\Processes\pipes-testchild.py

```
import os, time, sys
mypid     = os.getpid()
parentpid = os.getppid()
sys.stderr.write('Child %d of %d got arg: %s\n' %
                            (mypid, parentpid, sys.argv[1]))
for i in range(2):
    time.sleep(3)           # родительский процесс вводится в ожидание
    input = raw_input()     # stdin привязан к каналу: данные поступают
                            # из родительского stdout
    time.sleep(3)
    reply = 'Child %d got: [%s]' % (mypid, input)
    print reply             # stdout привязан к каналу: данные поступают на родительский stdin
    sys.stdout.flush()      # обеспечить отправку, иначе произойдет блокировка
```

Ниже тест показан в действии под Linux; чтение его выдачи не производит большого впечатления, но показывает, как две программы выполняются независимо и передают данные туда-сюда через устройство канала, управляемое операционной системой. Это еще более похоже на модель клиент/сервер (если представить себе дочерний процесс как сервер). Текст этой выдачи, заключенный в квадратные скобки, попал из родительского процесса в дочерний и вернулся обратно в родительский – и все это через каналы, соединенные со стандартными потоками:

```
[mark@toy]$ python pipes.py
Child 797 of 796 got arg: spam
Parent got: "Child 797 got: [Hello 1 from parent 796]"
Parent got: "Child 797 got: [Hello 2 from parent 796]"
```

Взаимная блокировка, сброс буфера и небуферизованные потоки

Эти два процесса ведут простой диалог, но его достаточно, чтобы проиллюстрировать некоторые опасности, таящиеся в процедурах обмена данными между программами. Во-первых, отметим, что обе программы должны выводить сообщения на stderr – их потоки stdout привязаны к входным потокам другой программы. Поскольку процессы используют общие дескрипторы файлов, устройство stderr одно и то же для родителя и его потомка, поэтому сообщения о состоянии выводятся в одно и то же место.

Более тонкая особенность состоит в том, что и родительский, и дочерний процессы после вывода текста в поток stdout вызывают sys.stdout.flush. Запрос ввода из канала обычно блокирует вызвавшего, если данных в канале нет, но в нашем примере из-за этого не должно возникать проблем: запись производится столько же раз, сколько чтение на другом конце канала. Однако по умолчанию sys.stdout *буферизуется*, поэтому выведенный текст в действительности может оказаться переданным только через некоторое время (когда будут заполнены до конца буферы вывода stdio). Факти-

чески, если не делать сброса буфера, оба процесса могут зависнуть в ожидании данных друг от друга – входных данных, находящихся в буфере и не сбрасываемых в канал. Это приводит к состоянию взаимной блокировки (*deadlock*), когда оба процесса блокируются в вызове `raw_input` и ожидают события, которое никогда не произойдет.

Имейте в виду, что буферизация выходных данных является в действительности функцией файловой системы, используемой для доступа к каналам, а не самих каналов (каналы помещают выходные данные в очередь, но не скрывают их от чтения!). На самом деле это происходит в данном примере только потому, что информация канала копируется в `sys.stdout` – встроенный файловый объект, по умолчанию использующий буферизацию `stdio`. Однако такие аномалии могут происходить и при использовании других средств коммуникации между процессами, например вызовов `popen2` и `popen3`, о которых говорилось в главе 2.

В целом, когда программы ведут такого рода двусторонний диалог, есть не менее трех способов избежать проблем взаимной блокировки, связанных с буферизацией:

- Как показано в данном примере, сброс выходных потоков в канал с помощью метода файла `flush` является простым способом принудительной очистки буферов.
- Допускается использование каналов в *небуферизованном режиме* – используйте вызовы низкого уровня модуля `os` для непосредственного чтения и записи по описателям каналов или (в большинстве систем) передайте в `os.fdopen` аргумент размера буфера, равный 0, чтобы отключить буферизацию `stdio` в файловом объекте, являющемся оболочкой для дескриптора. Для fifo, которые описываются в следующем разделе, то же самое нужно сделать при вызове `open`.
- Просто используйте флаг `-u` в командной строке Python, чтобы отключить буферизацию для потока `sys.stdout`.

Нужно сказать несколько слов о последнем приеме. Попробуйте сделать следующее: удалите все вызовы `sys.stdout.flush` в обоих примерах, 3.17 и 3.18 (файлы *pipes.py* и *pipes-testchild.py*), и измените в родительском *pipes.py* вызов, порождающий дочерний процесс, на следующий (т. е. добавьте аргумент командной строки `-u`):

```
spawn('python', '-u', 'pipes-testchild.py', 'spam')
```

После этого запустите программу с помощью командной строки `python -u pipes.py`. Работа будет происходить так же, как при вызове сброса `stdout` вручную, потому что `stdout` будет действовать в *небуферизованном* режиме. В целом, однако, взаимная блокировка представляет собой более обширную проблему, для обращения к которой здесь недостаточно места; с другой стороны, если у вас достаточно знаний, чтобы пытаться осуществлять IPC на Python, то, наверное, вы уже ветеран войн со взаимными блокировками.

Именованные каналы (fifo)

На некоторых платформах можно создавать каналы, существующие в виде файлов. Такие файлы называются «именованными каналами» (named pipes), или «fifo», так как они ведут себя в точности как каналы, которые создавались в предыдущих программах, но связаны с реальными файлами, располагающимися на компьютере и являющимися внешними для любой программы. После создания файла именованного канала процессы осуществляют чтение и запись в него с использованием обычных файловых операций. Fifo являются однонаправленными потоками, но группу из двух fifo можно использовать для осуществления двусторонней связи точно так же, как это делалось для анонимных каналов в предыдущем разделе.

Так как fifo являются файлами, они живут дольше, чем внутрипроцессные каналы, и к ним могут обращаться программы, запускаемые независимо. Приводившиеся выше примеры неименованных внутрипроцессных каналов основывались на том факте, что дескрипторы файлов (в том числе каналов) копируются в дочерние процессы. С помощью же fifo доступ к каналам производится по имени файла, которое видят все программы независимо от отношений между родительскими и дочерними процессами. Поэтому они лучше подходят в качестве механизма IPC для независимых программ клиента и сервера; например, постоянно выполняющаяся программа сервера может создавать fifo и ждать из них запросы, поступающие от произвольных клиентов, а не только ответвленных сервером.

В Python файлы именованных каналов создаются с помощью вызова `os.mkfifo`, который доступен в настоящее время только на Unix-подобных платформах и в Windows NT (но не в Windows 95/98). Однако этот вызов только создает внешний файл; для отправки и получения данных через fifo его нужно открыть и обрабатывать как стандартный файл. Пример 3.19 создан на основе приведенного выше сценария *pipe2.py*, переписанный для использования fifo вместо анонимных каналов.

Пример 3.19. PP2E\System\Processes\pipefifo.py

```
##########################################################################################
# именованные каналы; os.mkfifo недоступен в Windows 95/98; нет необходимости
# ветвления, поскольку файловые каналы fifo являются внешними для процессов,
# общие файловые дескрипторы не нужны;
##########################################################################################
import os, time, sys
fifoname = '/tmp/pipefifo'                          # должны открываться с одинаковыми именами

def child():
    pipeout = os.open(fifoname, os.O_WRONLY)        # открыть файл канала fifo как дескриптор
    zzz = 0
    while 1:
        time.sleep(zzz)
        os.write(pipeout, 'Spam %03d\n' % zzz)
        zzz = (zzz+1) % 5

def parent():
    pipein = open(fifoname, 'r')                    # открыть fifo как объект stdio
    while 1:
        line = pipein.readline()[:-1]               # блокируется до отправки данных
        print 'Parent %d got "%s" at %s' % (os.getpid(), line, time.time())

if __name__ == '__main__':
    if not os.path.exists(fifoname):
        os.mkfifo(fifoname)                         # создать файл именованного канала
    if len(sys.argv) == 1:

        parent()                                   # запуск как родителя, если нет аргументов args
    else:                                          # или как дочерний процесс
        child()
```

Поскольку fifo существует независимо и от родительского, и от дочернего процессов, нет причин делать ветвление: дочерний процесс может быть запущен независимо от родительского и должен лишь открыть файл fifo с таким же именем. Здесь, например, под Linux родитель запущен в одном окне xterm, а потомок затем в другом. Сообщения начинают появляться в окне родителя только после запуска потомка:

```
[mark@toy]$ python pipefifo.py
Parent 657 got "Spam 000" at 968390065.865
```

```
Parent 657 got "Spam 001" at 968390066.865
Parent 657 got "Spam 002" at 968390068.865
Parent 657 got "Spam 003" at 968390071.865
Parent 657 got "Spam 004" at 968390075.865
Parent 657 got "Spam 000" at 968390075.867
Parent 657 got "Spam 001" at 968390076.865
Parent 657 got "Spam 002" at 968390078.865

[mark@toy]$ file /tmp/pipefifo
/tmp/pipefifo: fifo (named pipe)
[mark@toy]$ python pipefifo.py -child
```

Сигналы

Сигналы, в отсутствие лучшей аналогии, можно сравнить с палкой, которой тыкают в процесс. Программы генерируют сигналы, чтобы сработал обработчик данного сигнала в другом процессе. Операционная система тоже этим занимается – некоторые сигналы, генерируемые при необычных системных событиях, могут убить программу, если их не обработать. Если это несколько напоминает возбуждение исключительных ситуаций в Python, то так оно и есть: сигналы являются программно генерируемыми событиями и аналогом исключительных ситуаций, действующим между процессами. Однако, в отличие от исключительных ситуаций, сигналы идентифицируются по номеру, не помещаются в очередь и в действительности являются механизмом *асинхронных событий*, управляемым операционной системой, вне области действия интерпретатора Python.

Для того чтобы сделать сигналы доступными в сценариях, Python предоставляет модуль `signal`, который позволяет регистрировать функции Python в качестве обработчиков событий сигналов. Этот модуль есть как на Unix-подобных платформах, так и под Windows (хотя в версии для Windows определено меньше перехватываемых сигналов). Для иллюстрации базового интерфейса сигналов сценарий примера 3.20 устанавливает функцию Python обработчика сигнала, номер которого передается как аргумент командной строки.

Пример 3.20. PP2E\System\Processes\signal1.py

```
##############################################################################
# перехват сигналов в Python; передать номер сигнала N как аргумент командной строки,
# использовать команду оболочки "kill -N pid" для передачи процессу сигнала; большинство
# обработчиков сигналов восстанавливается Python'ом после перехвата(смотрите в главе
# по сетевым сценариям детали SIGCHLD); модуль signal есть в Windows, но там определяется
# лишь несколько типов сигналов;
##############################################################################
import sys, signal, time
def now(): return time.ctime(time.time())      # строка с текущим временем
def onSignal(signum, stackframe):              # обработчик сигнала python
    print 'Got signal', signum, 'at', now()    # большинство обработчиков остается действующими
signum = int(sys.argv[1])
signal.signal(signum, onSignal)                  # установить обработчик сигнала
while 1: signal.pause()                          # ждать сигнала (или: pass)
```

Здесь действуют только два вызова из модуля `signal`:

- `signal.signal` принимает номер сигнала и объект функции и устанавливает эту функцию в качестве обработчика сигнала с данным номером при его возбуждении. Python автоматически восстанавливает большинство обработчиков сигналов, когда они возникают, поэтому нет необходимости повторно вызывать эту функцию

внутри самого обработчика сигнала, чтобы заново его зарегистрировать. За исключением SIGCHLD, обработчик сигнала остается установленным, пока не будет явно сброшен (например, путем установки его SIG_DFL, чтобы восстановить режим по умолчанию, или в SIG_IGN, чтобы игнорировать сигнал). Режим действия SIGCHLD зависит от платформы.

- signal.pause заставляет процесс уснуть, пока не будет перехвачен следующий сигнал. Вызов time.sleep аналогичен, но не работает с сигналами на моей машине Linux – он генерирует ошибку прерванного системного вызова. Цикл while 1: pass тоже остановит сценарий, но будет напрасно тратить ресурсы процессора.

Вот как действует этот сценарий при запуске под Linux: номер ожидаемого сигнала (12) передается в командной строке, а программа запускается в фоновом режиме с помощью оператора оболочки & (имеющегося в большинстве Unix-подобных оболочек):

```
[mark@toy]$ python signal1.py 12 &
[1] 809
[mark@toy]$ ps
  PID TTY          TIME CMD
  578 ttyp1    00:00:00 tcsh
  809 ttyp1    00:00:00 python
  810 ttyp1    00:00:00 ps
[mark@toy]$ kill -12 809
[mark@toy]$ Got signal 12 at Fri Sep  8 00:27:01 2000
kill -12 809
[mark@toy]$ Got signal 12 at Fri Sep  8 00:27:03 2000
kill -12 809
[mark@toy]$ Got signal 12 at Fri Sep  8 00:27:04 2000

[mark@toy]$ kill -9 809           # signal 9 always kills the process
```

Ввод и вывод здесь несколько перемешаны, потому что процесс осуществляет вывод на тот же экран, в котором вводятся новые команды оболочки. Для передачи программе сигнала команда оболочки *kill* принимает номер сигнала и ID процесса, которому он должен быть передан (809); всякий раз, когда очередная команда *kill* посылает сигнал, процесс отвечает сообщением, сгенерированным функцией Python обработчика сигнала.

Модуль signal экспортирует также функцию signal.alarm, с помощью которой задается интервал времени в секундах, по истечении которого должен быть отправлен сигнал SIGALRM. Чтобы вызывать и перехватывать тайм-ауты, установите таймер и обработчик SIGALRM, как в примере 3.21.

Пример 3.21. PP2E\System\Processes\signal2.py

```
##############################################################################
# установка и перехват сигналов тайм-аута в Python; time.sleep не очень хорошо
# работает с alarm (и сигналами вообще на моей Linux PC), поэтому
# вызывайте signal.pause, чтобы ничего не делалось до получения сигнала;
##############################################################################

import sys, signal, time
def now(): return time.ctime(time.time())

def onSignal(signum, stackframe):                # обработчик сигналов python
    print 'Got alarm', signum, 'at', now()       # большинство обработчиков остается
                                                 # действующими

while 1:
    print 'Setting at', now()
```

```
signal.signal(signal.SIGALRM, onSignal)      # установить обработчик сигнала
signal.alarm(5)                              # послать сигнал через 5 секунд
signal.pause()                               # ждать сигнал
```

При выполнении этого сценария под Linux функция обработчика `onSignal` вызывается каждые пять секунд:

```
[mark@toy]$ python signal2.py
Setting at Fri Sep  8 00:27:53 2000
Got alarm 14 at Fri Sep  8 00:27:58 2000
Setting at Fri Sep  8 00:27:58 2000
Got alarm 14 at Fri Sep  8 00:28:03 2000
Setting at Fri Sep  8 00:28:03 2000
Got alarm 14 at Fri Sep  8 00:28:08 2000
Setting at Fri Sep  8 00:28:08 2000
```

Вообще говоря, сигналы следует использовать осторожно, что не явствует из приведенных примеров. В частности, некоторые системные вызовы плохо реагируют на прерывание сигналами, а в многопоточной программе только главный поток может устанавливать обработчики сигналов и реагировать на них.

Однако при правильном использовании сигналы предоставляют механизм связи, основанный на событиях. Он не такой мощный, как потоки данных типа каналов, но в некоторых ситуациях его достаточно, например, когда нужно только сообщить программе, что произошло нечто важное, не передавая подробностей о самом событии. Иногда сигналы сочетают с другими средствами IPC. Например, начальный сигнал может сообщить программе, что клиент хочет установить связь через именованный канал – примерно как похлопать кого-то по плечу, чтобы привлечь его внимание, прежде чем начать говорить. На большинстве платформ резервируется один или несколько номеров сигналов `SIGUSR` для определяемых пользователем событий такого рода.

Запуск программ под Windows

Представьте себе на мгновение, что вам предложили написать большую книгу по Python, и вы хотите сделать так, чтобы читатель мог легко запускать примеры из книги на любой платформе, где выполняется Python. Книга – это хорошо, но очень приятно иметь возможность сразу щелкнуть по демонстрационной программе. Стало быть, требуется написать на Python общую и переносимую программу для запуска других программ Python. Как это сделать?

В данной главе мы уже видели, как переносимым образом порождать потоки, но они являются лишь параллельно выполняемыми функциями, а не внешними программами. Мы научились также запускать новые независимо выполняющиеся программы – как с помощью комбинации `fork/exec`, так и средств для запуска команд оболочки, например `os.popen`. Однако попутно я несколько раз отмечал, что вызов `os.fork` в настоящее время не работает в Windows, а `os.popen` дает отказ при вызове в Python версии 1.5.2 и более ранних при вызове из программ GUI под Windows; эти ограничения могут быть сняты к тому моменту, когда вы читаете эту книгу (например, версия 2.0 улучшает `os.popen` в Windows), но когда я ее писал, этого еще не произошло. Более того, по причинам, которые будут объяснены позже, вызов `os.popen` в некоторых сценариях имеет тенденцию к блокированию (остановке) вызвавшего его.

К счастью, в стандартной библиотеке Python есть другие способы запуска программ, хотя и специфические для различных платформ:

- Вызовы `os.spawnv` и `os.spawnve` запускают программы в Windows сходным с комбинацией `fork/exec` на Unix-подобных платформах образом.

- Вызов os.system можно использовать в Windows для запуска команды DOS *start*, которая независимо открывает (т. е. выполняет) файл, основываясь на связи между именами файлов и программами в Windows, как если бы по нему был сделан щелчок.
- Средства, предоставляемые пакетом расширения Python win32all, предоставляют другие, менее стандартные способы запуска программ (например, вызов WinExec).

Вызов os.spawnv

Из всех этих способов наиболее сложен вызов spawnv, но, вместе с тем, он наиболее похож на ветвление программ в Unix. В действительности он не копирует вызывающий процесс (поэтому операции, использующие общие описатели, не работают), но может использоваться для запуска программы Windows, выполняемой совершенно независимо от вызвавшей. Сценарий примера 3.22 делает сходство более очевидным: он запускает программу с помощью комбинации fork/exec в Linux или вызова os.spawnv в Windows.

Пример 3.22. PP2E\System\Processes\spawnv.py

```
########################################################################
# запустить параллельно 10 экземпляров child.py; запускать программы
# в Windows через spawnv (как fork+exec). P_OVERLAY замещает,
# P_DETACH отправляет дочерний stdout в никуда
########################################################################
import os, sys

for i in range(10):
    if sys.platform[:3] == 'win':
        pypath  = r'C:\program files\python\python.exe'
        os.spawnv(os.P_NOWAIT, pypath, ('python', 'child.py', str(i)))
    else:
        pid = os.fork()
        if pid != 0:
            print 'Process %d spawned' % pid
        else:
            os.execlp('python', 'python', 'child.py', str(i))
print 'Main process exiting.'
```

Вызывайте os.spawnv с флагом режима процесса, полным путем к интерпретатору Python и набором строк, представляющих командную строку DOS, которая должна запускать новую программу. Флаг *process mode* определяется в Visual C++ (библиотека которого обеспечивает работу вызова spawnv); обычно используются следующие значения:

- P_OVERLAY – порождаемая программа замещает вызывающую, как в os.exec
- P_DETACH – запускает полностью независимую программу, без ожидания
- P_NOWAIT – запускает программу без ожидания выхода из нее; возвращает ее дескриптор
- P_WAIT – запускает программу и останавливается до ее завершения; возвращает код ее завершения

Выполните вызов dir(os), чтобы посмотреть, какие еще флаги есть, и либо выполните несколько тестов, либо посмотрите дополнительные подробности в документации VC++; такие вещи, как политика соединения стандартных потоков, имеют тонкие различия между режимами P_DETACH и P_NOWAIT. Вот действие этого сценария под Win-

dows, порождающее 10 независимых экземпляров программы Python *child.py*, с которой мы познакомились ранее в этой главе:

```
C:\...\PP2E\System\Processes>type child.py
import os, sys
print 'Hello from child', os.getpid(), sys.argv[1]
C:\...\PP2E\System\Processes>python spawnv.py
Hello from child -583587 0
Hello from child -558199 2
Hello from child -586755 1
Hello from child -562171 3
Main process exiting.
Hello from child -581867 6
Hello from child -588651 5
Hello from child -568247 4
Hello from child -563527 7
Hello from child -543163 9
Hello from child -587083 8
```

Обратите внимание, что эти копии программы производят свою выдачу в случайном порядке, а родительская программа завершается раньше, чем завершатся все дочерние; все эти программы действительно выполняются в Windows параллельно. Заметьте также, что выдача дочерней программы появляется в окне консоли, где была запущена *spawnv.py*; при использовании `P_NOWAIT` стандартный выход попадает на родительскую консоль, но никуда не попадает, если использовать флаг `P_DETACH` (что не является ошибкой при порождении программ GUI).

Вызов `os.spawnve` работает так же, как `os.spawnv`, но принимает дополнительный четвертый аргумент со словарем, в котором задается свое окружение оболочки для порожденной программы (по умолчанию наследующее все установки родителя).

Выполнение командных строк DOS

Вызовы `os.system` и `os.popen` можно использовать для выполнения командных строк в Windows так же, как в Unix-подобных платформах (но с упомянутыми выше предостережениями относительно `popen`). Однако в Windows команда DOS *start* в сочетании с `os.system` дает простой способ сценариям запускать любые файлы в системе с использованием связи имен файлов с программами в Windows. Запуск файла программы таким способом заставляет ее выполняться так же независимо, как запустившая ее программа. Пример 3.23 демонстрирует такую технику запуска.

Пример 3.23. PP2E\System\Processes\dosstart.py

```
########################################################################
# запускает параллельно 5 экземпляров child.py;
# - в Windows os.system всегда блокирует вызвавшего,
#   а os.popen в данное время не работает в программах GUI
# - использование команды DOS start вызывает окно DOS (которое
#   исчезает сразу после завершения программы child.py)
# - при запуске child-wait.py с помощью DOS start появляются 5
#   независимых окон консоли DOS, которые остаются (1 на программу)
# команда DOS start использует связь имен файлов при запуске с файлом Python, как при двойном
# щелчке в проводнике Windows (таким способом можно запустить файл с любым именем);
########################################################################
import os, sys
for i in range(5):
    #print os.popen('python child.py ' + str(i)).read()[:-1]
    #os.system('python child.py ' + str(i))
```

```
    #os.system('start child.py ' + str(i))
     os.system('start child-wait.py ' + str(i))
print 'Main process exiting.'
```

Раскомментируйте одну из строк этого сценария в цикле `for`, чтобы поэкспериментировать с этими схемами на своем компьютере. На моей машине при запуске сценария с раскомментированием одного из первых двух вызовов я получаю выдачу такого вида – текст, выведенный пятью порожденными программами Python:

```
C:\...\PP2E\System\Processes>python dosstart.py
Hello from child -582331 0
Hello from child -547703 1
Hello from child -547703 2
Hello from child -547651 3
Hello from child -547651 4
Main process exiting.
```

Вызов `os.system` обычно блокирует вызвавшего, пока не произойдет завершение порожденной программы; чтение данных, выводимых вызовом `os.popen`, имеет такой же блокирующий эффект (читающая программа ждет завершения вывода порожденной программы). Но при раскомментировании одного из последних двух утверждений в цикле я получаю всего лишь такую выдачу:

```
C:\...\PP2E\System\Processes>python dosstart.py
Main process exiting.
```

В обоих случаях на дисплее видны также пять новых совершенно независимых окон консоли DOS; при раскомментировании третьей строки в цикле все окна DOS уходят сразу после своего появления; когда действует последняя строка в цикле, они остаются на экране после выхода из программы `dosstart`, потому что сценарий `child-wait` останавливается для ввода перед завершением.

Использование команды DOS start

Чтобы понять, почему это происходит, нужно сначала разобраться, как действует команда DOS *start* в целом. Грубо говоря, командная строка DOS вида `start` *command* действует, как если бы *command* вводилась в диалоговом окне Windows «Run», которое можно открыть через меню кнопки Start. Если *command* является именем файла, он открывается точно так же, как если щелкнуть по его имени в графическом интерфейсе селектора файлов Windows Explorer.

Например, следующие три команды DOS автоматически запускают с файлом *index.html* Internet Explorer, с *uk-1.jpg* – зарегистрированную у меня программу просмотра графических изображений и с *sousa.au* – мою программу проигрывания звуковых файлов. Windows просто открывает файл в той программе, которая определена для обработки файлов указанного формата. Более того, все три эти программы выполняются независимо от того окна консоли DOS, в котором введена команда:

```
C:\temp>start c:\stuff\website\public_html\index.html
C:\temp>start c:\stuff\website\public_html\uk-1.jpg
C:\...\PP2E\System\Processes>start ..\..\Internet\Ftp\sousa.au
```

Теперь, поскольку команда *start* может запускать любой файл и командную строку, нет причин, по которым ее нельзя использовать для запуска независимо выполняемой программы Python:

```
C:\...\PP2E\System\Processes>start child.py 1
```

Поскольку при установке Python регистрируется для открытия имен, оканчивающихся на *.py*, это сработает – сценарий *child.py* будет запущен независимо от окна консоли DOS, несмотря на то что не было задано имя или путь к программе интерпретатора Python. Однако поскольку *child.py* просто выводит сообщение и завершается, результат не вполне удовлетворяет: новое окно DOS появляется, чтобы обслуживать стандартный вывод сценария, и тут же исчезает, когда завершается дочерняя программа (это та особенность «вспышек» Windows, о которой говорилось выше!). Лучше будет, если добавить в конец программы вызов `raw_input`, чтобы перед завершением происходило ожидание нажатия какой-либо клавиши:

```
C:\...\PP2E\System\Processes>type child-wait.py
import os, sys
print 'Hello from child', os.getpid(), sys.argv[1]
raw_input("Press <Enter>")   # don't flash on Windows
C:\...\PP2E\System\Processes>start child-wait.py 2
```

Теперь окно DOS дочерней программы всплывает и сохраняется после возврата из команды *start*. Нажатие клавиши Enter во всплывающем окне DOS заставляет его закрыться.

Использование start в сценариях Python

Так как мы знаем, что вызовы Python `os.system` и `os.popen` можно выполнять в сценарии для запуска *любой* командной строки, которую можно ввести в ответ на приглашение оболочки DOS, можно запускать из сценария Python независимо выполняемые программы просто выполнением командной строки DOS *start*. Например:

```
C:\...\PP2E>python
>>> import os
>>>
>>> cmd = r'start c:\stuff\website\public_html\index.html'  # запустить броузер IE
>>> os.system(cmd)                                          # выполняется независимо
0
>>> file = r'gui\gifs\pythonPowered.gif'                 # запустить средство просмотра графики
>>> os.system('start ' + file)                           # IE откроет .gif
0
>>> os.system('start ' + 'Gui/gifs/PythonPowered.gif')     # прямая косая черта тоже действует
0
>>> os.system(r'start Internet\Ftp\sousa.au')         # запустить проигрыватель мультимедиа
0
```

Четыре представленные здесь вызова `os.system` запускают броузер веб-страниц, средство просмотра графики и проигрыватель звуковых файлов, которые зарегистрированы на машине для открытия файлов *.html*, *.gif* и *.au* (если только эти программы уже не выполняются). Запущенные программы выполняются совершенно независимо от сеанса Python при выполнении команды DOS start вызов `os.system` не ждет завершения запущенной программы. Например, на рис. 3.1 показана работа обработчика файлов *.gif*, установленного на моей машине, вызываемая вторым и третьим вызовами `os.system` в предшествующем коде.

Теперь, так как мы знаем, что программу Python можно запустить из командной строки, получаем два способа запуска программ Python:

```
C:\...\PP2E>python
>>> os.system(r'python Gui\TextEditor\textEditor.pyw')   # запустить и ждать
0
>>> os.system(r'start  Gui\TextEditor\textEditor.pyw')   # запустить и продолжить
0
```

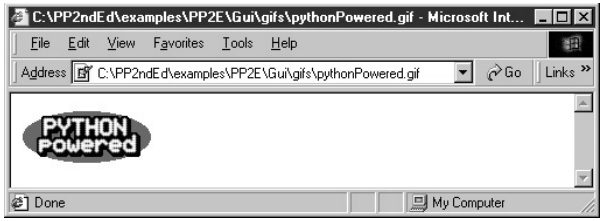


Рис. 3.1. Запущенное средство просмотра графики (Internet Explorer)

При выполнении команды `python` вызов `os.system` ждет завершения команды (блокирует). При выполнении команды *start* этого не происходит – запущенная программа Python (в данном случае PyEdit, GUI текстового редактора, с которым мы познакомимся в главе 9 «Более крупные приложения GUI») выполняется независимо от вызвавшего ее `os.system`. И наконец, это причина, по которой следующий вызов в *dosstart.py* создает новый независимый экземпляр *child-wait.py*:

```
C:\...\PP2E\System\Processes>python
>>> os.system('start child-wait.py 1')
0
```

При запуске этот вызов влечет появление нового независимого окна консоли DOS, которое должно служить стандартными входным и выходным потоками программы child-wait. Оно действительно независимо; на самом деле оно будет работать, даже если завершить данный сеанс интерпретатора Python и окно консоли DOS, в котором была введена команда.[1] Вызов `os.popen` может запускать команду *start* тоже; но поскольку он обычно и так запускает команды независимо, единственными видимыми здесь преимуществами *start* являются всплывающее окно DOS и то, что не требуется нахождение Python в системном пути поиска:

```
>>> file = os.popen('start child-wait.py 1')    # против: python child-wait...
>>> file.read()
'Hello from child -413849 1\012Press <Enter>'
```

Какую же схему тогда использовать? Вызов `os.system` или `os.popen` для запуска команды `python` работает отлично, но только если пользователь добавил каталог *python.exe* в системный путь поиска. Выполнение команды DOS *start* часто оказывается более простой альтернативой выполнению команд `python` или вызову функции `os.spawnv`, так как связь с именем файла автоматически устанавливается вместе с Python, а `os.spawnv` требует указания полного пути к каталогу с интерпретатором Python (*python.exe*). С другой стороны, запуск команд *start* с помощью вызова `os.system` может оказаться неудачным в Windows, если командная строка слишком длинная:

```
>>> os.system('start child-wait.py ' + 'Z'*425)     # OK - 425 Z в окне dos
```

[1] И запомните, что если вам нужно запустить таким способом программу Python GUI и вообще не показывать новое окно DOS стандартных потоков, просто дайте сценарию имя *child-wait.pyw*; «w» в конце сообщает версии Python для Windows, что окно DOS не нужно. Для любителей DOS – у команды *start* есть несколько интересных опций: `/m` (выполнять свернутым), `/max` (выполнять в полноэкранном режиме), `/r` (выполнять в восстановленном размере – по умолчанию) и `/w` (не возвращаться до завершения запущенной программы – этим при необходимости можно заблокировать вызвавшего). Для получения подсказки введите `start /?`. А для разработчиков Unix, пытающихся подглядывать через забор, совет: вы тоже можете запускать независимые программы с помощью `os.system` – добавьте в командную строку оператор `&` выполнения в фоновом режиме.

```
0
>>> os.system('start child-wait.py ' + 'Z'*450)     # неудача - сообщение,
                                                    # не исключительная ситуация
Access is denied.
0
>>> os.popen('python child-wait.py ' + 'Z'*500).read()    # работает, если настроена PATH
>>> os.system('python child-wait.py ' + 'Z'*500)          # работает, если настроена PATH

>>> pypath = r'C:\program files\python\python.exe'        # тоже работает
>>> os.spawnv(os.P_NOWAIT, pypath, ('python', 'child-wait.py', 'Z'*500))
```

Практическое правило рекомендует использовать `os.spawnv`, если команды длинные (или могут быть длинными). Например, в главе 4 «Более крупные системные примеры, часть 1», встретится сценарий, запускающий веб-броузер для просмотра файлов HTML; хотя команда *start*, примененная к файлу HTML, автоматически запустит программу броузера, в этом сценарии нужно использовать вместо нее `os.spawnv` для работы с файлами HTML, которые могут иметь длинные пути.

Дополнительные сведения о других специфических для Windows средствах запуска программ можно найти в книге издательства O'Reilly «Python Programming on Win32». Другие существующие схемы еще менее стандартны, чем приведенные здесь, но отлично освещены в той книге.

Структура переносимого запуска программ

Из-за различия в запуске программ на разных платформах может оказаться трудным запомнить, какие средства должны использоваться в конкретной ситуации. Более того, некоторые из этих средств вызываются способами, которые настолько сложны, что быстро забываются (мной, по крайней мере). Я пишу сценарии, которым требуется запускать программы Python, настолько часто, что в итоге создал особый модуль, постаравшись скрыть в нем большую часть подспудных деталей. Работая над этим модулем, я сделал его достаточно сообразительным, чтобы он мог автоматически выбирать схему запуска, соответствующую платформе, на которой он применяется. Лень породила не один полезный модуль.

В примере 3.24 в один модуль собрана немалая часть тех приемов, которые встретились нам в этой главе. В нем реализован абстрактный суперкласс `LaunchMode`, определяющий, что значит запустить программу Python, но не определяющий, как это сделать. Вместо этого его подклассы предоставляют метод `run`, который действительно запускает программу Python согласно выбранной схеме, и (по выбору) определяют метод `announce` для вывода имени программы при запуске.

Пример 3.24. PP2E\launchmodes.py

```
###############################################################
# запуск программ Python с помощью классов схем запуска;
# предполагает нахождение 'python' в системном пути (но смотрите Launcher.py)
###############################################################

import sys, os, string
pycmd = 'python'   # предполагается, что он в системном пути

class LaunchMode:
    def __init__(self, label, command):
        self.what  = label
        self.where = command
    def __call__(self):                  # при вызове, например: функция обратного
                                         # вызова, срабатывающая по нажатию кнопки
```

```
        self.announce(self.what)
        self.run(self.where)              # подкласс должен определять run()
    def announce(self, text):             # подкласс может переопределять метод
        print text                        # announce() вместо логики if/elif
    def run(self, cmdline):
        assert 0, 'run must be defined'

class System(LaunchMode):                             # выполнение команд оболочки
    def run(self, cmdline):                           # осторожно: блокирует вызвавшего,
        os.system('%s %s' % (pycmd, cmdline))         # если не добавить '&', в Linux

class Popen(LaunchMode):                              # осторожно: блокирует вызвавшего,
    def run(self, cmdline):                           # так как канал закрывается слишком скоро
        os.popen(pycmd + ' ' + cmdline)               # 1.5.2 не работает в Windows GUI

class Fork(LaunchMode):
    def run(self, cmdline):
        assert hasattr(os, 'fork')                    # сегодня для linux/unix
        cmdline = string.split(cmdline)               # преобразовать строку в список
        if os.fork() == 0:                            # запустить новый дочерний процесс
            os.execvp(pycmd, [pycmd] + cmdline)       # запустить новую программу
                                                      # в дочернем процессе

class Start(LaunchMode):
    def run(self, cmdline):                     # только для Windows
        assert sys.platform[:3] == 'win'        # выполняется независимо от вызвавшего
        os.system('start ' + cmdline)           # использует связи Windows

class Spawn(LaunchMode):                        # только для Windows
    def run(self, cmdline):                     # запуск python в новом процессе
        assert sys.platform[:3] == 'win'        # выполняется независимо от вызвавшего
       #pypath = r'C:\program files\python\python.exe'
        try:                                                # получить путь к python
            pypath = os.environ['PP2E_PYTHON_FILE']         # запущен стартером?
        except KeyError:                                    # если да - настроить среду
            from Launcher import which, guessLocation
            pypath = which('python.exe', 0) or guessLocation('python.exe', 1,0)
        os.spawnv(os.P_DETACH, pypath, ('python', cmdline)) # P_NOWAIT: окно dos

class Top_level(LaunchMode):
    def run(self, cmdline):                          # новое окно, тот же процесс
        assert 0, 'Sorry - mode not yet implemented'  # доделать: нужна информация о классе GUI

if sys.platform[:3] == 'win':
    PortableLauncher = Spawn                         # выбор лучшего стартера для платформы
else:                                                # этот код нужно изменить в других местах
    PortableLauncher = Fork

class QuietPortableLauncher(PortableLauncher):
    def announce(self, text):
        pass

def selftest():
    myfile  = 'launchmodes.py'
    program = 'Gui/TextEditor/textEditor.pyw ' + myfile       # предполагается в cwd
    raw_input('default mode...')
    launcher = PortableLauncher('PyEdit', program)
    launcher()                                                # не блокирует

    raw_input('system mode...')
    System('PyEdit', program)()                               # блокирует
```

```
    raw_input('popen mode...')
    Popen('PyEdit', program)()                          # блокирует

    if sys.platform[:3] == 'win':
        raw_input('DOS start mode...')                  # не блокирует
        Start('PyEdit', program)()

if __name__ == '__main__': selftest()
```

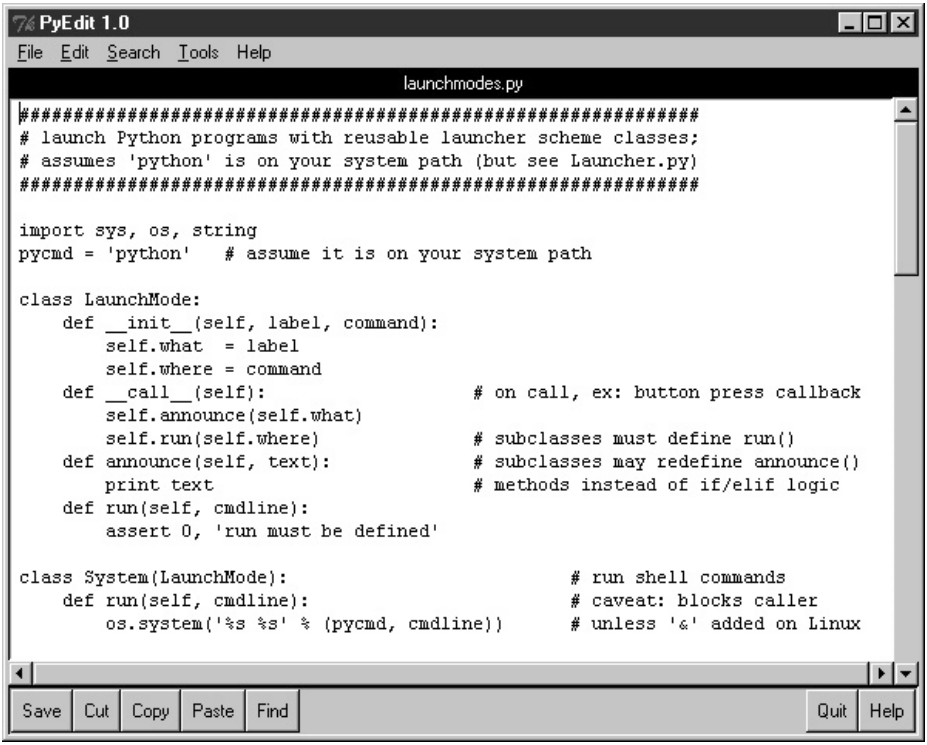
Ближе к концу файла модуль выбирает класс по умолчанию исходя из атрибута `sys.platform`: в Windows `PortableLauncher` устанавливается в класс, использующий `spawnv`, и в класс, использующий комбинацию `fork/exec`, на других платформах. Если импортировать этот модуль и всегда использовать его атрибут `PortableLauncher`, то можно позабыть о многочисленных специфических для платформы деталях, перечисленных в данной главе.

Чтобы запустить программу Python, просто импортируйте класс `PortableLauncher`, создайте экземпляр, передав метку и командную строку (без слова «python» впереди), а затем вызовите объект экземпляра, как если бы это была функция. Программа запускается операцией *call* вместо метода, чтобы классы этого модуля можно было использовать для создания обработчиков обратного вызова в GUI, использующих Tkinter. Как будет показано в следующих главах, нажатие кнопок в Tkinter запускает вызываемый объект без аргументов; зарегистрировав экземпляр `PortableLauncher` для обработки нажатия кнопки, можно автоматически запускать новую программу из GUI другой программы.

При автономном выполнении, как обычно, вызывается функция этого модуля `selftest`. И в Windows и в Linux все тестируемые классы запускают программу текстового редактора на Python (снова предстоящая GUI-программа PyEdit), выполняющуюся независимо в собственном окне. На рис. 3.2 она показана в работе под Windows; все порождаемые редакторы автоматически открывают файл исходного кода *launchmodes.py*, так как его имя передается PyEdit как аргумент командной строки. В соответствии с кодом `System` и `Popen` блокируют вызвавшего до выхода из редактора, а `PortableLauncher` (в действительности `Spawn` или `Fork`) и `Start` — нет:[1]

```
C:\...\PP2E>python launchmodes.py
default mode...
PyEdit
system mode...
PyEdit
popen mode...
PyEdit
DOS start mode...
PyEdit
```

[1] Это довольно тонкий момент. Технически `Popen` блокирует вызвавшего только потому, что входной канал в порожденную программу закрывается слишком рано, когда результат вызова `os.popen` удаляется сборщиком мусора в `Popen.run`; `os.popen` обычно не блокирует (в действительности присвоение его результата глобальной переменной временно откладывает блокировку, но только до того, как очередной запуск объекта `Popen` удалит предыдущий результат). В Linux добавление `&` в конец создаваемой командной строки в методах `System` и `Popen.run` приводит к тому, что эти объекты больше не блокируют вызвавшие их программы при своем выполнении. Поскольку схемы `fork/exec`, `spawnv` и `system/start` действуют на практике ничуть не хуже, этими состояниями блокировки в `Popen` разработчики не занимались. Обратите также внимание, что схема `Start` не создает всплывающего окна консоли DOS при самотестировании лишь потому, что имя файла программы текстового редактора имеет расширение *.pyw*; запуск файлов программ *.py* с помощью `os.system` обычно создает всплывающее окно консоли.




```
############################################################
# launch Python programs with reusable launcher scheme classes;
# assumes 'python' is on your system path (but see Launcher.py)
############################################################

import sys, os, string
pycmd = 'python'   # assume it is on your system path

class LaunchMode:
    def __init__(self, label, command):
        self.what  = label
        self.where = command
    def __call__(self):                    # on call, ex: button press callback
        self.announce(self.what)
        self.run(self.where)               # subclasses must define run()
    def announce(self, text):              # subclasses may redefine announce()
        print text                         # methods instead of if/elif logic
    def run(self, cmdline):
        assert 0, 'run must be defined'

class System(LaunchMode):                              # run shell commands
    def run(self, cmdline):                            # caveat: blocks caller
        os.system('%s %s' % (pycmd, cmdline))          # unless '&' added on Linux
```

Рис. 3.2. Программа PyEdit, запущенная с помощью launchmodes

Практическое применение этот файл находит в сценариях запуска, разработанных для выполнения переносимым образом примеров из этой книги. Сценарии PyDemos и PyGadgets в вершине дерева примеров этой книги (смотрите прилагаемый CD) просто импортируют `PortableLauncher` и регистрируют экземпляры, чтобы отвечать на события GUI. Благодаря этому оба GUI запускающих программ работают в Windows и Linux без изменений (конечно, в этом помогает и переносимость Tkinter). Сценарий PyGadgets даже настраивает `PortableLauncher` для изменения метки в GUI во время запуска.

```
class Launcher(launchmodes.PortableLauncher):    # использовать оболочку для класса launcher
    def announce(self, text):                     # настройка для установки метки GUI
        Info.config(text=text)
```

Мы исследуем эти сценарии в части II «Программирование для GUI» (но можно прямо сейчас заглянуть в конец главы 8 «Обзор Tkinter, часть II»). Из-за этой роли класс `Spawn` в данном файле использует дополнительные средства для поиска пути к исполняемому файлу Python, который требуется `os.spawnv`. Он вызывает две функции, экспортируемые файлом *Launcher.py*, для поиска подходящего *python.exe* независимо от того, добавил ли пользователь его каталог в системную переменную PATH. Идея в том, чтобы запускать программы Python, даже если Python не прописан в переменных оболочки на локальной машине. Однако, поскольку мы еще познакомимся с *Launcher.py* в главе 4, я намерен отложить пока дальнейшее изучение деталей.

Другие системные средства

В этой и предыдущих главах мы познакомились с большинством обычно используемых инструментов из библиотеки Python. Попутно мы научились использовать их для таких полезных вещей, как запуск программ, обработка каталогов и т. д. Следующие две главы отчасти служат продолжением этой темы – в них инструменты, с которыми мы только что познакомились, используются для реализации сценариев, выполняющих более полезную и практическую работу на системном уровне, так что продолжение этой истории следует.

Тем не менее, в Python есть и другие системные средства, которые появятся в этой книге дальше. Например:

- *Сокеты* (используемые для связи с другими программами и по сетям) представляются в главе 10.
- Вызовы *Select* (используемые для организации многозадачности) также представлены в главе 10 как средство реализации серверов.
- Вызовы *блокировки файлов* из модуля `fcntl` появятся в главе 14.
- *Регулярные выражения* (поиск строк по шаблону, используемый во многих средствах обработки текста) появятся только в главе 18 «Текст и языки».

Кроме того, такие вещи, как *ветвление* и *потоки*, интенсивно используются в главах, посвященных сценариям для Интернета: смотрите реализации серверов в главе 10 и графические интерфейсы FTP и электронной почты в главе 11. В действительности большая часть инструментов, описанных в этой главе, будет постоянно появляться в дальнейших примерах этой книги – чего еще можно ожидать от переносимых библиотек общего назначения?

Последнее, но не самое маловажное, что я хотел бы еще раз подчеркнуть: в библиотеке Python есть много других инструментов, которые вообще не фигурируют в данной книге, – при наличии около 200 библиотечных модулей авторам, пишущим книгу по Python, приходится быть сдержанными в отборе тем! Как всегда, напомню о необходимости изучения руководств по библиотеке Python в начале и на всем протяжении вашей карьеры программиста Python.

4

Более крупные системные примеры, часть 1

«Расколы, объединения и вторжения чужих»

В этой главе и следующей за ней мы продолжим рассмотрение области *системных утилит* в Python. В них собраны более значительные сценарии Python, выполняющие реальную системную работу – сравнение и копирование деревьев каталогов, расщепление файлов, поиск файлов и каталогов, тестирование других программ, настройка окружения оболочек программ, запуск веб-броузеров и т. д. Для облегчения усвоения это собрание разделено на две главы. В данной главе представлены разнообразные системные утилиты на Python, иллюстрирующие типичные задачи и приемы, используемые в этой области. В следующей главе находятся более крупные программы Python, нацеленные на более сложную обработку файлов и каталогов.

Хотя главной целью этих двух глав, посвященных изучению конкретных примеров, является дать вам возможность получить представление о реалистичных сценариях в работе, размер этих примеров дает возможность на практике посмотреть на поддержку в Python таких основ разработки, как ООП и повторное использование кода. Лишь в контексте таких нетривиальных программ, с которыми мы здесь познакомимся, эти средства начинают приносить ощутимые плоды. В этих главах подчеркивается также «замысел» системных средств, а не только способ их реализации, – попутно я укажу на задачи реального мира, которые решаются изучаемыми нами примерами, чтобы помочь вам увидеть детали в их контексте.

Предварительное замечание: в этих главах изложение ведется быстро, и некоторые примеры состоят в основном из листингов, предназначенных для самостоятельного изучения. Поскольку все представленные сценарии подробно документированы и используют системные средства Python, описанные в двух предыдущих главах, я не стану подробно разбирать код. Вам следует прочесть листинги исходного кода и поэкспериментировать с этими программами на своем компьютере, чтобы лучше почувствовать, как системные интерфейсы объединяются для выполнения реальных задач. Исходный код всех примеров есть на прилагаемом к книге CD-ROM, и большинство из них работает на всех основных платформах.

Должен также заметить, что это те программы, которыми я реально пользуюсь, а не примеры, написанные специально для книги. В действительности они писались на протяжении нескольких лет и выполняют весьма различающиеся задачи, поэтому их трудно связать единой нитью. С другой стороны, они, в первую очередь, способствуют разъяснению пользы системных средств, демонстрируют концепции крупных разработок, чего нельзя сделать на более простых примерах, и в совокупности свидетельствуют в пользу простоты и переносимости автоматизации системных задач с помощью Python. Когда вы овладеете основами, то пожалеете, что не сделали этого раньше.

Разрезание и соединение файлов

Мои дети, как и многие другие, проводят массу времени в Интернете. Насколько мне известно, в наши дни это считается стоящим занятием. Среди этого нового поколения компьютерные пахари и гуру пользуются таким же уважением, как когда-то рок-звезды среди моего. Когда дети скрываются в своих комнатах, они, скорее всего, возятся с компьютерами, а не осваивают гитарные рифы. Пожалуй, это более здоровое занятие, чем некоторые утехи моей собственной растраченной юности, но это уже тема для книги иного сорта.

Но если у вас есть дети-подростки и компьютеры или они есть у ваших знакомых, то мысль последить за тем, чем эти подростки занимаются в Сети, вам, вероятно, покажется правильной. Введите свое любимое нецензурное слово практически в любой поисковый механизм веб, и вы поймете причину этой озабоченности – эта информация значительно лучше той, которую мне удавалось добыть в годы своего отрочества. Из-за этой проблемы только несколько компьютеров в моем доме имеют выход в Интернет.

Когда мои дети выходят на одну из этих машин, они скачивают массу игр. Однако чтобы избежать заражения наших Очень Важных Компьютеров вирусами, детям обычно приходится загружать игры на компьютер с выходом в Интернет, а затем переносить их на свои компьютеры и там устанавливать. Проблема в том, что файлы с играми не маленькие: обычно они слишком велики для гибкого диска (а запись на CD отнимает ценное игровое время).

Если бы на всех машинах в моем доме стояла Linux, проблем бы не было. В Unix есть стандартные программы командной строки для разрезания файла на достаточно маленькие для гибкого диска кусочки (*split*), в то время как другие программы собирают эти кусочки вместе и воссоздают исходный файл (*cat*). Однако поскольку у нас в доме стоят самые разные машины, требуется более переносимое решение.

Разрезание файлов переносимым способом

Так как на всех компьютерах в моем доме стоит Python, на помощь приходит простой переносимый сценарий на нем. Программа Python примера 4.1 распределяет содержимое одного файла по группе частичных файлов, которые записывает в каталог.

Пример 4.1. PP2E\System\Filetools\split.py

```
#!/usr/bin/python
###############################################################################
# разрезать файл на несколько частей; потом можно соединить их вместе с помощью join.py;
# это пользовательская версия split - стандартной утилиты командной строки unix; так как она
# написана на Python, то работает и под Windows, и ее легко подправить; так как она
# экспортирует функцию, ее легко импортировать и использовать в других приложениях;
###############################################################################

import sys, os
kilobytes = 1024
megabytes = kilobytes * 1000
chunksize = int(1.4 * megabytes)                    # по умолчанию примерно размер дискеты

def split(fromfile, todir, chunksize=chunksize):
    if not os.path.exists(todir):                   # ошибки обрабатывает вызвавший
        os.mkdir(todir)                             # создать каталог для чтения/записи кусков
    else:
        for fname in os.listdir(todir):             # удалить существующие файлы
            os.remove(os.path.join(todir, fname))
```

```
    partnum = 0
    input = open(fromfile, 'rb')                    # в Windows использовать двоичный режим
    while 1:                                        # eof=прочтена пустая строка
        chunk = input.read(chunksize)               # получить новый кусок <= chunksize
        if not chunk: break
        partnum  = partnum+1
        filename = os.path.join(todir, ('part%04d' % partnum))
        fileobj  = open(filename, 'wb')
        fileobj.write(chunk)
        fileobj.close()                             # или просто open().write()
    input.close()
    assert partnum <= 9999                          # сортировка join не получится, если 5 цифр
    return partnum

if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == '-help':
        print 'Use: split.py [file-to-split target-dir [chunksize]]'
    else:
        if len(sys.argv) < 3:
            interactive = 1
            fromfile = raw_input('File to be split? ')          # ввод при нажатии кнопки
            todir    = raw_input('Directory to store part files? ')
        else:
            interactive = 0
            fromfile, todir = sys.argv[1:3]                     # аргументы командной строки
            if len(sys.argv) == 4: chunksize = int(sys.argv[3])
        absfrom, absto = map(os.path.abspath, [fromfile, todir])
        print 'Splitting', absfrom, 'to', absto, 'by', chunksize

        try:
            parts = split(fromfile, todir, chunksize)
        except:
            print 'Error during split:'
            print sys.exc_type, sys.exc_value
        else:
            print 'Split finished:', parts, 'parts are in', absto
        if interactive: raw_input('Press Enter key') # пауза
```

По умолчанию этот сценарий разрезает входной файл на куски, примерно равные размеру дискеты – идеально для перемещения больших файлов между не связанными между собой машинами. Самое важное, что поскольку это полностью переносимый код, данный сценарий будет работать почти везде, даже там, где нет своей встроенной программы для разрезания файлов. Все что требуется – иметь установленный Python. Вот действие этого сценария при разрезании самоустанавливающегося исполняемого модуля Python 1.5.2 под Windows:

```
C:\temp>echo %X%              сокращенная запись переменной окружения
C:\PP2ndEd\examples\PP2E

C:\temp>ls -l py152.exe
-rwxrwxrwa   1 0        0        5028339 Apr 16  1999 py152.exe

C:\temp>python %X%\System\Filetools\split.py -help
Use: split.py [file-to-split target-dir [chunksize]]

C:\temp>python %X%\System\Filetools\split.py py152.exe pysplit
Splitting C:\temp\py152.exe to C:\temp\pysplit by 1433600
Split finished: 4 parts are in C:\temp\pysplit

C:\temp>ls -l pysplit
```

```
total 9821
-rwxrwxrwa   1 0        0        1433600 Sep 12 06:03 part0001
-rwxrwxrwa   1 0        0        1433600 Sep 12 06:03 part0002
-rwxrwxrwa   1 0        0        1433600 Sep 12 06:03 part0003
-rwxrwxrwa   1 0        0         727539 Sep 12 06:03 part0004
```

Каждый из четырех созданных здесь файлов частей представляет один двоичный кусок файла *py152.exe*, достаточно маленький, чтобы разместиться на одной дискете. Действительно, если сложить вместе размеры созданных частей, показанные командой *ls*, то получится 5 028 339 байт – ровно столько, каков размер исходного файла. Прежде чем пытаться снова сложить вместе эти файлы, рассмотрим некоторые более тонкие места разрезающего сценария.

Режимы работы

Данный сценарий может получать свои параметры в *интерактивном* режиме или режиме *командной строки*; он проверяет количество аргументов в командной строке, чтобы узнать, в каком режиме используется. В режиме командной строки указываются файл, который нужно разрезать, и выходной каталог; при этом с помощью необязательного третьего аргумента можно переопределить установленный по умолчанию размер части.

В интерактивном режиме сценарий запрашивает имя файла и выходной каталог в окне консоли с помощью `raw_input` и перед завершением делает остановку, ожидая нажатия клавиши. Этот режим удобен, когда файл программы запускается щелчком по его значку – в Windows параметры вводятся во всплывающее окно DOS, которое не исчезает автоматически. Сценарий также показывает абсолютные пути для своих параметров (пропуская их через `os.path.abspath`), потому что в интерактивном режиме они могут не быть очевидны. Чуть позже мы рассмотрим примеры использования других режимов разрезания.

Двоичный доступ к файлам

Этот код достаточно осторожен, чтобы открывать входные и выходные файлы в двоичном режиме (`rb`, `wb`), потому что он должен переносимым образом обрабатывать такие файлы, как исполняемые или аудио, а не только текст. В главе 2 «Системные инструменты» мы узнали, что в Windows текстовые файлы автоматически отображают символы конца строки `\r\n` в `\n` при вводе и `\n` в `\r\n` при выводе. Необходимо, чтобы в данных, действительно являющихся двоичными, символы `\r` не исчезали при чтении, а при записи в выдачу не попадали лишние символы `\r`. Для файлов, открываемых в двоичном режиме, такая трансформация `\r` подавляется при выполнении сценария под Windows, и искажения данных не происходит.

Закрытие файлов вручную

Этот сценарий заботится о том, чтобы вручную закрыть используемые им файлы. Например:

```
fileobj = open(partname, 'wb')
fileobj.write(chunk)
fileobj.close()
```

Как тоже было показано в главе 2, эти три строки обычно можно заменить одной:

```
open(partname, 'wb').write(chunk)
```

Эта более короткая форма использует то обстоятельство, что в текущей реализации Python файлы автоматически закрываются при уничтожении объектов файлов (то есть при уборке их как мусора, когда не остается ссылок на объект файла). В этой строке объект файла будет уничтожен немедленно, потому что результат `open` является в выражении временным и ссылка на него не сохраняется в каком-либо более долго существующем имени. Аналогичным образом при выходе из функции `split` уничтожается объект файла `input`.

Однако когда я писал эту главу, существовала вероятность того, что такой режим автоматического закрытия в будущем может исчезнуть.[1] Более того, в JPython – основанной на Java-реализации Python – объекты, на которые нет ссылок, не уничтожаются с такой поспешностью, как в стандартном Python. Если сейчас или в будущем вам может потребоваться перенос на Java, а ваш сценарий в состоянии создать много файлов за короткий промежуток времени и, возможно, будет выполняться на машине, которая ограничивает количество открытых файлов в каждой программе, выполняйте закрытие вручную. Для моих задач вызывать `close` в этом сценарии никогда не требовалось, но поскольку функция разрезания в этом модуле задумана как средство общего назначения, учтены варианты такого наихудшего развития событий.

Соединение файлов переносимым образом

Вернемся к перемещению больших файлов по дому. После загрузки больших файлов игровых программ мои дети обычно выполняют предшествующий сценарий для разрезания, щелкнув по его названию в Windows Explorer и введя имена файлов. После разрезания они просто копируют каждую часть файла на дискету, идут с дискетами наверх и воссоздают каталог с разрезанным файлом на нужном компьютере, копируя файлы с дискет. Затем щелкают по сценарию примера 4.2 или запускают его другим способом, чтобы вновь соединить части.

Пример 4.2. PP2E\System\Filetools\join.py

```
#!/usr/bin/python
###############################################################################
# соединить вместе частичные файлы, созданные split.py. Это примерно соответствует
# команде 'cat fromdir/* > tofile' в unix, но лучше переносится и настраивается,
# а также экспортирует операцию соединения как многократно используемую функцию.
# Зависит от порядка сортировки имен файлов: они должны быть одной длины.
# Можно расширить, чтобы split/join выводили селекторы файлов из Tkinter.
###############################################################################

import os, sys
readsize = 1024

def join(fromdir, tofile):
    output = open(tofile, 'wb')
    parts  = os.listdir(fromdir)
    parts.sort()
    for filename in parts:
        filepath = os.path.join(fromdir, filename)
        fileobj  = open(filepath, 'rb')
```

[1] Надеюсь, что этого не произойдет – такое изменение было бы существенным отходом от обратной совместимости, что повлияло бы на системы под управлением Python по всему миру. С другой стороны, это лишь возможное направление мутаций Python в будущем. Мне говорили, что издатели технических книг любят, когда в языках происходят изменения, а эта книга не касается политики.

```
        while 1:
            filebytes = fileobj.read(readsize)
            if not filebytes: break
            output.write(filebytes)
        fileobj.close()
    output.close()

if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == '-help':
        print 'Use: join.py [from-dir-name to-file-name]'
    else:
        if len(sys.argv) != 3:
            interactive = 1
            fromdir = raw_input('Directory containing part files? ')
            tofile  = raw_input('Name of file to be recreated? ')
        else:
            interactive = 0
            fromdir, tofile = sys.argv[1:]
        absfrom, absto = map(os.path.abspath, [fromdir, tofile])
        print 'Joining', absfrom, 'to make', absto

        try:
            join(fromdir, tofile)
        except:
            print 'Error joining files:'
            print sys.exc_type, sys.exc_value
        else:
           print 'Join complete: see', absto
        if interactive: raw_input('Press Enter key')    # pause if clicked
```

После выполнения сценария `join` детям может потребоваться выполнить что-то еще вроде `zip`, `gzip` или `tar`, чтобы распаковать архивный файл, если он поставлялся не исполняемым;[1] но в любом случае они значительно ближе к тому, чтобы увидеть космический корабль Enterprise в действии. Вот как происходит объединение файлов в Windows, сливающее вместе только что созданные нами куски файлов:

```
C:\temp>python %X%\System\Filetools\join.py -help
Use: join.py [from-dir-name to-file-name]

C:\temp>python %X%\System\Filetools\join.py pysplit mypy152.exe
Joining C:\temp\pysplit to make C:\temp\mypy152.exe
Join complete: see C:\temp\mypy152.exe

C:\temp>ls -l mypy152.exe py152.exe
-rwxrwxrwa   1 0        0        5028339 Sep 12 06:05 mypy152.exe
-rwxrwxrwa   1 0        0        5028339 Apr 16  1999 py152.exe

C:\temp>fc /b mypy152.exe py152.exe
Comparing files mypy152.exe and py152.exe
FC: no differences encountered
```

[1] Смотрите также встроенный модуль *gzip.py* в стандартной библиотеке Python; он предоставляет средства для чтения и записи файлов `gzip`, имена которых обычно имеют расширение `.gz`. Его можно использовать для распаковки сжатых `gzip` файлов, и он служит в Python общим эквивалентом стандартных утилит командной строки `gzip` и `gunzip`. Этот встроенный модуль в свою очередь использует модуль `zlib`, в котором реализовано `gzip`-совместимое сжатие данных. В Python 2.0 есть также новый модуль `zipfile` для обработки архивов в формате ZIP (отличающемся от `gzip`).

Сценарий join использует os.listdir, чтобы собрать все части файла в каталоге, созданном при разрезании, и сортирует список имен файлов, чтобы составить части вместе в правильном порядке. Получается точная байт-в-байт копия исходного файла (что проверяется выше командой DOS *fc*; под Unix используйте *cmp*).

Конечно, часть этого процесса выполняется вручную (я еще не придумал, как запрограммировать этап «идут с дискетами наверх»), но с помощью сценариев split и join перемещение больших файлов становится быстрым и простым. Так как этот сценарий является также переносимым кодом Python, он выполняется на любой платформе, на которую может понадобиться перенести разрезанные файлы. Например, мои дети загружают игры как для Windows, так и для Linux; и так как сценарий выполняется на любой из платформ, у них нет проблем.

Чтение файлов блоками целиком

Прежде чем двинуться дальше, следует отметить несколько деталей в коде сценария соединения. Прежде всего, обратите внимание, что сценарий работает с файлами в двоичном режиме, а также читает каждый частичный файл блоками размером в 1 Кбайт. В действительности значение readsize (размер блоков, читаемых из входного частичного файла) не имеет никакого отношения к chunksize в *split.py* (общий размер каждого выходного файла). Как было показано в главе 2, каждый частичный файл можно было бы прочесть сразу целиком:

```
filebytes = open(filepath, 'rb').read()
output.write(filebytes)
```

Недостаток такой схемы в том, что в ней действительно в оперативную память загружается сразу весь файл. Например, при чтении частичного файла размером 1,4 Мбайт в память целиком в ней создается строка размером 1,4 Мбайт, содержащая все байты файла. Поскольку split разрешает пользователям задавать даже более крупный размер куска, сценарий join ожидает худшего и читает блоками ограниченного размера. Полная надежность была бы обеспечена, если бы сценарий split также читал свои входные данные меньшими порциями, но на практике этого не потребовалось.

Сортировка имен файлов

Если внимательно изучить код сценария, то можно также заметить, что схема соединения полностью зависит от порядка сортировки имен файлов в каталоге с частичными файлами. Так как сценарий соединения просто вызывает метод sort для списка имен файлов, возвращаемого os.listdir, он подразумевает, что при разрезании создаются файлы с одинаковыми длиной и форматом имени. Разрезающий сценарий использует форматирующее выражение с дополнением незначащими нулями ('part%04d'), чтобы обеспечить в конце имен файлов присутствие одинакового количества цифр (четырех), подобно следующему списку:

```
>>> list = ['xx008', 'xx010', 'xx006', 'xx009', 'xx011', 'xx111']
>>> list.sort()
>>> list
['xx006', 'xx008', 'xx009', 'xx010', 'xx011', 'xx111']
```

Ведущие нули в маленьких числах гарантируют, что частичные файлы правильно сортируются для соединения. Если бы не было ведущих нулей, join неправильно работал бы, когда файл разрезается больше чем на девять частей, потому что верх берет первая цифра:

```
>>> list = ['xx8', 'xx10', 'xx6', 'xx9', 'xx11', 'xx111']
>>> list.sort()
```

```
>>> list
['xx10', 'xx11', 'xx111', 'xx6', 'xx8', 'xx9']
```

Так как метод `sort` принимает в качестве аргумента функцию сортировки, в принципе можно вырезать цифры из имен файлов и выполнить числовую сортировку:

```
>>> list = ['xx8', 'xx10', 'xx6', 'xx9', 'xx11', 'xx111']
>>> list.sort(lambda x, y: cmp(int(x[2:]), int(y[2:])))
>>> list
['xx6', 'xx8', 'xx9', 'xx10', 'xx11', 'xx111']
```

Но тогда требуется, чтобы все имена файлов начинались с подстроки одинаковой длины, поэтому зависимость между сценариями `split` и `join` сохраняется. Однако поскольку эти сценарии задуманы как два шага одного и того же процесса, какие-то зависимости между ними кажутся оправданными.

Варианты использования

Проделаем еще несколько экспериментов с этими системными утилитами Python, чтобы продемонстрировать другие режимы работы. Если аргументы заданы в командной строке не полностью, `split` и `join` достаточно сообразительны, чтобы вводить параметры *интерактивно*. Рассмотрим снова процесс разрезания и склеивания самоустанавливающегося файла Python под Windows, когда параметры вводятся в окне консоли DOS:

```
C:\temp>python %X%\System\Filetools\split.py
File to be split? py152.exe
Directory to store part files? splitout
Splitting C:\temp\py152.exe to C:\temp\splitout by 1433600
Split finished: 4 parts are in C:\temp\splitout
Press Enter key

C:\temp>python %X%\System\Filetools\join.py
Directory containing part files? splitout
Name of file to be recreated? newpy152.exe
Joining C:\temp\splitout to make C:\temp\newpy152.exe
Join complete: see C:\temp\newpy152.exe
Press Enter key

C:\temp>fc /B py152.exe newpy152.exe
Comparing files py152.exe and newpy152.exe
FC: no differences encountered
```

При двойном щелчке по файлам этих программ в GUI менеджера файлов они работают таким же образом (обычно при таком запуске нет аргументов командной строки). В таком режиме вывод абсолютного пути помогает прояснить, где в действительности находятся файлы. Помните, что при щелчке по файлу текущим рабочим каталогом является исходный каталог сценария, поэтому имя *tempsplit* в действительности находится в каталоге с исходным кодом; введите полный путь, чтобы поместить разрезанные части файла в другое место:

```
[во всплывающем окне консоли DOS после щелчка по split]
File to be split? c:\temp\py152.exe
Directory to store part files? tempsplit
Splitting c:\temp\py152.exe to C:\PP2ndEd\examples\PP2E\System\Filetools\
tempsplit by 1433600
Split finished: 4 parts are in C:\PP2ndEd\examples\PP2E\System\Filetools\
tempsplit
```

```
Press Enter key

[во всплывающем окне консоли DOS после щелчка по join]
Directory containing part files? tempsplit
Name of file to be recreated? c:\temp\morepy152.exe
Joining C:\PP2ndEd\examples\PP2E\System\Filetools\tempsplit to make
c:\temp\morepy152.exe
Join complete: see c:\temp\morepy152.exe
Press Enter key
```

Поскольку основная логика этих сценариев оформлена в виде функций, очень просто использовать их код, импортировав и вызвав из другого компонента Python:

```
C:\temp>python
>>> from PP2E.System.Filetools.split import split
>>> from PP2E.System.Filetools.join  import join
>>>
>>> numparts = split('py152.exe', 'calldir')
>>> numparts
4
>>> join('calldir', 'callpy152.exe')
>>>
>>> import os
>>> os.system(r'fc /B py152.exe callpy152.exe')
Comparing files py152.exe and callpy152.exe
FC: no differences encountered
0
```

Замечание относительно производительности. Все приведенные здесь проверки `split` и `join` обрабатывают файл размером 5 Мбайт, но завершаются не более чем за 1 секунду реального времени на моих переносных компьютерах 300 и 650 МГц под Windows 98 – достаточно быстро для любого мыслимого применения. (Они выполняются еще быстрее после кэширования Windows информации об используемых файлах.) Оба файла столь же быстро выполняются и с другими файлами разумных размеров; вот как файл разрезается на куски по 500 000 и 50 000 байт:

```
C:\temp>python %X%\System\Filetools\split.py py152.exe tempsplit 500000
Splitting C:\temp\py152.exe to C:\temp\tempsplit by 500000
Split finished: 11 parts are in C:\temp\tempsplit

C:\temp>ls -l tempsplit
total 9826
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0001
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0002
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0003
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0004
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0005

-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0006
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0007
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0008
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0009
-rwxrwxrwa   1 0        0         500000 Sep 12 06:29 part0010
-rwxrwxrwa   1 0        0          28339 Sep 12 06:29 part0011

C:\temp>python %X%\System\Filetools\split.py py152.exe tempsplit 50000
Splitting C:\temp\py152.exe to C:\temp\tempsplit by 50000
Split finished: 101 parts are in C:\temp\tempsplit
C:\temp>ls tempsplit
part0001  part0014  part0027  part0040  part0053  part0066  part0079  part0092
```

```
part0002  part0015  part0028  part0041  part0054  part0067  part0080  part0093
part0003  part0016  part0029  part0042  part0055  part0068  part0081  part0094
part0004  part0017  part0030  part0043  part0056  part0069  part0082  part0095
part0005  part0018  part0031  part0044  part0057  part0070  part0083  part0096
part0006  part0019  part0032  part0045  part0058  part0071  part0084  part0097
part0007  part0020  part0033  part0046  part0059  part0072  part0085  part0098
part0008  part0021  part0034  part0047  part0060  part0073  part0086  part0099
part0009  part0022  part0035  part0048  part0061  part0074  part0087  part0100
part0010  part0023  part0036  part0049  part0062  part0075  part0088  part0101
part0011  part0024  part0037  part0050  part0063  part0076  part0089
part0012  part0025  part0038  part0051  part0064  part0077  part0090
part0013  part0026  part0039  part0052  part0065  part0078  part0091
```

Разрезание может выполняться дольше, если размеры частичных файлов заданы настолько маленькими, что создаются тысячи частей – разрезание на 1006 частей выполняется медленнее (на моем компьютере такие разрезание и склеивание происходят за 5 и 2 секунды соответственно и связаны с тем, какие еще программы открыты):

```
C:\temp>python %X%\System\Filetools\split.py py152.exe tempsplit 5000
Splitting C:\temp\py152.exe to C:\temp\tempsplit by 5000
Split finished: 1006 parts are in C:\temp\tempsplit

C:\temp>python %X%\System\Filetools\join.py tempsplit mypy152.exe
Joining C:\temp\tempsplit to make C:\temp\py152.exe
Join complete: see C:\temp\py152.exe

C:\temp>fc /B py152.exe mypy152.exe
Comparing files py152.exe and mypy152.exe
FC: no differences encountered

C:\temp>ls -l tempsplit
... 1000 строк удалено ...
-rwxrwxrwa   1 0        0           5000 Sep 12 06:30 part1001
-rwxrwxrwa   1 0        0           5000 Sep 12 06:30 part1002
-rwxrwxrwa   1 0        0           5000 Sep 12 06:30 part1003
-rwxrwxrwa   1 0        0           5000 Sep 12 06:30 part1004
-rwxrwxrwa   1 0        0           5000 Sep 12 06:30 part1005
-rwxrwxrwa   1 0        0           3339 Sep 12 06:30 part1006
```

Наконец, разрезающий сценарий достаточно умен, чтобы создать выходной каталог, если его не существует, или очистить его от старых файлов, если он существует. Так как соединяющий сценарий склеивает все файлы, существующие в выходном каталоге, это полезная эргономическая функция: можно легко позабыть что в выходном каталоге находятся файлы от предыдущего прогона. Поскольку эти сценарии предназначены для выполнения моими детьми, они должны быть возможно более снисходительны; если даже контингент ваших пользователей будет иным, возможно, он будет не сильно отличаться.

```
C:\temp>python %X%\System\Filetools\split.py py152.exe tempsplit 700000
Splitting C:\temp\py152.exe to C:\temp\tempsplit by 700000
Split finished: 8 parts are in C:\temp\tempsplit

C:\temp>ls -l tempsplit
total 9827
-rwxrwxrwa   1 0        0         700000 Sep 12 06:32 part0001
-rwxrwxrwa   1 0        0         700000 Sep 12 06:32 part0002
-rwxrwxrwa   1 0        0         700000 Sep 12 06:32 part0003
...
... здесь только новые файлы ...
```

```
...
-rwxrwxrwa   1 0        0          700000 Sep 12 06:32 part0006
-rwxrwxrwa   1 0        0          700000 Sep 12 06:32 part0007
-rwxrwxrwa   1 0        0          128339 Sep 12 06:32 part0008
```

Создание веб-страниц со ссылками переадресации

Перемещаться всегда трудно, даже в прекрасном новом мире киберпространства. Изменение адреса вашей страницы в Интернете может привести к всякого рода неудобствам: приходится просить знакомых использовать новый адрес и надеяться, что остальные в конце концов наткнутся на него сами. Но если вы зависите от Интернета, то перемещение вызывает не меньше проблем, чем смена адреса в реальном мире.

К сожалению, таких перемещений сайта часто невозможно избежать. Как ISP (провайдеры интернет-услуг), так и серверы с течением времени приходят и уходят. Кроме того, некоторые провайдеры допускают падение уровня обслуживания до неприемлемого уровня; если вам не повезло и случилось подписаться на услуги такого провайдера, не остается ничего иного, как сменить его, а это часто требует изменения адреса в Сети.[1]

Представьте себе, однако, что вы пишете книги для O'Reilly и опубликовали свой адрес в Интернете во многих книгах, продаваемых по всему свету. Что делать, если качество обслуживания вашего ISP такое, что требуется переместить сайт? Оповещение об этом десятков или сотен тысяч читателей не представляется практичным решением.

Вероятно, лучшее, что можно сделать, это на достаточно продолжительный промежуток времени поместить на прежнем сайте инструкции по переадресации – виртуальный эквивалент вывески «Мы переехали по новому адресу» в витрине магазина. В Сети такое объявление может автоматически отправлять посетителей на новый сайт: нужно просто оставить на прежнем сайте страничку, содержащую гиперссылку на адрес страницы на новом сайте. При наличии таких *файлов со ссылками переадресации* посетители прежнего адреса окажутся на расстоянии одного щелчка от нового адреса.

Выглядит просто. Но поскольку посетители могут попытаться непосредственно обратиться по адресу *любого* файла на вашем прежнем сайте, вообще-то требуется оставить по одному файлу со ссылкой переадресации для каждого прежнего файла – страниц HTML, графических файлов и т. д. Если вам нравится бездумно вводить с клавиатуры большие объемы данных, можете вручную создать все файлы со ссылками переадресации. Но с учетом того, что мой домашний сайт в настоящее время содержит 140 файлов, перспектива запускать редактор для каждого файла оказалась более чем достаточной мотивацией для автоматизированного решения.

Файл шаблона страницы

Вот что я придумал. Прежде всего, я создаю текстовый файл общего *шаблона страницы*, показанный в примере 4.3 и описывающий вид всех файлов со ссылками переадресации, части которых будут заполнены позднее.

---

[1] Это случается. Действительно, большинство из тех, кто проводит в киберпространстве значительное время, может рассказать пару ужасных историй. Моя состоит в следующем: у меня был провайдер, который полностью отключился на несколько недель в результате проникновения в систему защиты со стороны бывшего служащего. Что еще хуже, персональная электронная почта не просто не работала, но накапливавшиеся сообщения были утеряны навсегда. Если ваше существование зависит от электронной почты и WWW в такой же мере, как мое, вы хорошо представляете, какую панику может вызвать подобное отключение.

Пример 4.3. PP2E\System\Filetools\template.html

```
<HTML><BODY>
<H1>This page has moved</H1>

<P>This page now lives at this address:

<P><A HREF="http://$server$/$home$/$file$">
http://$server$/$home$/$file$</A>

<P>Please click on the new address to jump to this page, and
update any links accordingly.
</P>

<HR>
<H3><A HREF="ispmove.html">Why the move? - The ISP story</A></H3>

</BODY></HTML>
```

Чтобы полностью разобраться в этом шаблоне, требуется некоторое знание HTML – языка описания веб-страниц, который мы рассмотрим в главе 12 «Сценарии на стороне сервера». Но для задач данного примера можно проигнорировать большую часть этого файла и сосредоточиться только на тех частях, которые окружены знаками доллара: строки `$server$`, `$home$` и `$file$` являются элементами, которые должны быть заменены реальными значениями с помощью глобальной подстановки текста. Эти элементы зависят от места, куда перемещен сайт, и файла.

Сценарий генератора страниц

При наличии файла шаблона страницы сценарий Python из примера 4.4 автоматически генерирует все необходимые файлы со ссылками переадресации.

Пример 4.4. PP2E\System\Filetools\site-forward.py

```
#########################################################################################
# Создать страницы со ссылками переадресации для перемещения веб-сайта. Для каждого
# существующего на сайте файла генерируется одна страница; загрузите сгенерированные
# страницы на свой прежний веб-сайт. Замечание по производительности: первые 2 вызова
# string.replace можно  вывести наружу из цикла for, но на моей машине Win98
# это выполняется <1 сек для 150 файлов сайта. Замечание по библиотеке: вызов os.listdir
# можно заменить следующим: sitefiles = glob.glob(sitefilesdir + os.sep + '*'), но тогда
# надо разделять имена файлов/каталогов с помощью: dirname, filename = os.path.split(sitefile);
#########################################################################################
import os, string
servername   = 'starship.python.net'     # адрес, куда перемещается сайт
homedir      = '~lutz/home'              # корень сайта
sitefilesdir = 'public_html'             # локальный адрес файлов сайта
uploaddir    = 'isp-forward'             # где хранить файлы переадресации
templatename = 'template.html'           # шаблон генерируемых страниц
try:
    os.mkdir(uploaddir)                  # при необходимости создать каталог для отправки
except OSError: pass
template  = open(templatename).read()    # загрузить или импортировать текст шаблона
sitefiles = os.listdir(sitefilesdir)     # имена файлов без префикса каталога
count = 0
for filename in sitefiles:
    fwdname = os.path.join(uploaddir, filename)                    # или + os.sep + filename
    print 'creating', filename, 'as', fwdname
    filetext = string.replace(template, '$server$', servername)   # вставить текст
```

```
        filetext = string.replace(filetext, '$home$',   homedir)      # и записать
        filetext = string.replace(filetext, '$file$',   filename)     # измененный файл
        open(fwdname, 'w').write(filetext)
        count = count + 1
    print 'Last file =>\n', filetext
    print 'Done:', count, 'forward files created.'
```

Обратите внимание, что текст шаблона загружается путем чтения *файла*; можно было бы и закодировать его как импортируемую строковую переменную Python (например, строку в тройных кавычках в файле модуля). Заметьте также, что все параметры конфигурации задаются присваиваниями в начале *сценария*, а не аргументами командной строки; поскольку они меняются очень редко, удобнее просто один раз напечатать их в самом сценарии.

Но главное, что нужно отметить – этому сценарию совершенно безразлично, как выглядит файл шаблона: он просто слепо выполняет в его тексте глобальные подстановки с различными именами файлов для каждого генерируемого файла. На самом деле можно как угодно изменить файл шаблона и не касаться при этом сценария. Такое разделение труда может быть использовано в любом типе контекста – при генерации make-файлов, писем-бланков и т. д. В отношении библиотечных средств сценарий генератора просто делает следующее:

- Использует `os.listdir` для обхода всех имен файлов в каталоге сайта
- Использует `string.replace` для осуществления операций глобального поиска и замены, заполняющих элементы текста файла шаблона, ограниченные символами `$`
- Использует `os.path.join` и встроенные объекты файлов для записи полученного текста в файл со ссылками переадресации с тем же именем в выходном каталоге

Окончательным результатом является зеркальное отражение первоначального каталога веб-сайта, содержащее только файлы со ссылками переадресации, созданные по шаблону страницы. Дополнительным преимуществом сценария генератора является возможность выполнения практически на любой платформе Python – у меня он выполняется на переносном компьютере под Windows (где хранятся файлы моего веб-сайта), а также на Unix-сервере, где хранится копия моего сайта. Вот как он выполняется под Windows:

```
C:\Stuff\Website>python %X%\System\Filetools\site-forward.py
creating about-hopl.html as isp-forward\about-hopl.html
creating about-lp-toc.html as isp-forward\about-lp-toc.html
creating about-lp.html as isp-forward\about-lp.html
creating about-pp-japan.html as isp-forward\about-pp-japan.html
...
... здесь удалены строки ...
...
creating whatsold.html as isp-forward\whatsold.html
creating xlate-lp.html as isp-forward\xlate-lp.html
creating about-pp2e.html as isp-forward\about-pp2e.html
creating about-ppr2e.html as isp-forward\about-ppr2e.html
Last file =>
<HTML><BODY>
<H1>This page has moved</H1>
<P>This page now lives at this address:
<P><A HREF="http://starship.python.net/~lutz/home/about-ppr2e.html">
http://starship.python.net/~lutz/home/about-ppr2e.html</A>
<P>Please click on the new address to jump to this page, and
update any links accordingly.
</P>
```

```
<HR>
<H3><A HREF="ispmove.html">Why the move? - The ISP story</A></H3>
</BODY></HTML>
Done: 137 forward files created.
```

Для проверки выдачи этого сценария сделайте двойной щелчок по любому из выходных файлов и посмотрите, как он выглядит в веб-броузере (или выполните команду *start* в консоли DOS в Windows, например `start isp-forward\about-ppr2e.html`). На рис. 4.1 показано, как выглядит одна из сгенерированных страниц на моей машине.

Для завершения процесса еще необходимо установить ссылки переадресации: загрузите все сгенерированные файлы из выходного каталога в веб-каталог вашего старого сайта. Если и это слишком большой объем для ручной работы, посмотрите, как это можно сделать автоматически с помощью Python посредством сценария загрузки на сервер по FTP в главе 11 «Сценарии на стороне клиента» (это выполняет *PP2E\Internet\Ftp\uploadflat.py*). Заразившись вирусом написания сценариев, вы поразитесь тому, какой объем ручного труда можно автоматизировать с помощью Python.

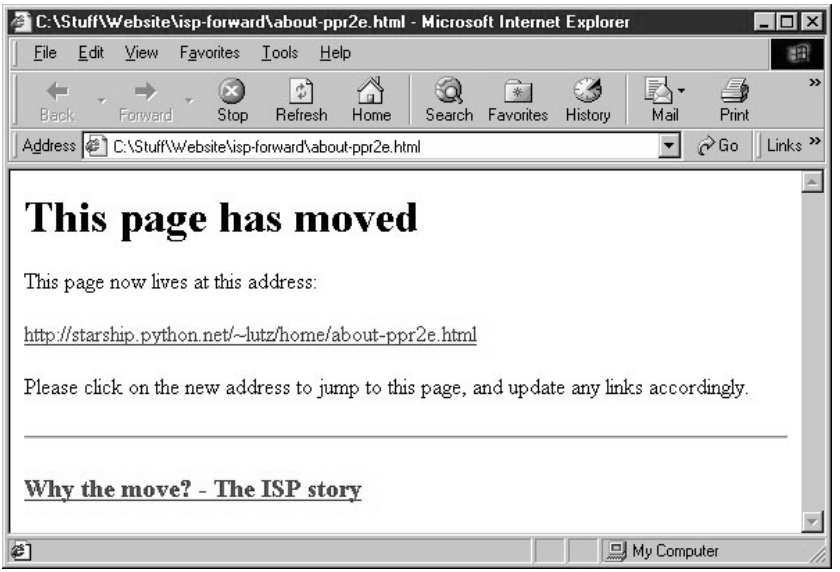


Рис. 4.1. Страница выходного файла переадресации сайта

Сценарий регрессивного теста

Как мы уже видели, Python предоставляет интерфейсы для целого ряда системных служб, а также инструменты для добавления новых. В примере 4.5 показаны некоторые часто используемые интерфейсы в действии. В нем реализована простая система *регрессивного тестирования*, выполняющая программу командной строки с заданным набором входных файлов и сравнивающая выдачу каждого прогона с предыдущим результатом. Этот сценарий был адаптирован для этой книги из автоматизированной тестирующей системы, которую я написал, чтобы отлавливать ошибки, появляющиеся в результате изменений в исходных файлах программы: в большой системе нельзя быть уверенным в том, что исправление не является в действительности скрытой ошибкой.

Пример 4.5. PP2E\System\Filetools\regtest.py

```
#!/usr/local/bin/python
import os, sys                          # получить сервисы unix в python
from stat import ST_SIZE                # или использовать os.path.getsize
from glob import glob                   # доп. возможности работы с именами файлов
from os.path import exists              # проверка существования файла
from time import time, ctime            # функции времени

print 'RegTest start.'
print 'user:', os.environ['USER']       # переменные окружения
print 'path:', os.getcwd()              # текущий каталог
print 'time:', ctime(time()), '\n'
program = sys.argv[1]                   # два аргумента командной строки
testdir = sys.argv[2]

for test in glob(testdir + '/*.in'):    # для всех подходящих входных файлов
    if not exists('%s.out' % test):
        # нет предыдущих результатов
        os.system('%s < %s > %s.out 2>&1' % (program, test, test))
        print 'GENERATED:', test
    else:
        # создать резервную копию, запустить, сравнить
        os.rename(test + '.out', test + '.out.bkp')
        os.system('%s < %s > %s.out 2>&1' % (program, test, test))
        os.system('diff %s.out %s.out.bkp > %s.diffs' % ((test,)*3) )
        if os.stat(test + '.diffs')[ST_SIZE] == 0:
            print 'PASSED:', test
            os.remove(test + '.diffs')
        else:
            print 'FAILED:', test, '(see %s.diffs)' % test

print 'RegTest done:', ctime(time())
```

Часть сценария имеет уклон в сторону Unix. Например, синтаксис `2>&1` для переадресации `stderr` действует в Unix и Windows NT/2000, но не в Windows 9x, а порождаемая командная строка *diff* является утилитой Unix. Такой код приходится немного исправлять, чтобы он мог выполняться на некоторых платформах. Кроме того, усовершенствования, внесенные в вызовы `popen` модуля `os` в Python 2.0, делают их более переносимым способом переадресации потоков в таких сценариях и альтернативой для синтаксиса переадресации команд оболочки.

Но в основе идея этого сценария проста: для каждого имени файла с суффиксом *.in* из каталога теста этот сценарий выполняет программу, указанную в командной строке, и ищет отклонения в ее результатах. Это простой способ выявить изменения (называемые «регрессиями») в поведении программ, вызываемых из оболочки. Настоящий секрет успеха этого сценария лежит в именах файлов, используемых для записи информации тестирования: внутри заданного каталога тестирования *testdir*.

- Файлы `testdir/`*`test`*`.in` представляют источники стандартного ввода для прогонов программы.
- Файлы `testdir/`*`test`*`.in.out` представляют выдачу, генерируемую для каждого входного файла.
- Файлы `testdir/`*`test`*`.in.out.bkp` являются резервными копиями предыдущих файлов результатов *.in.out*.
- Файлы `testdir/`*`test`*`.in.diffs` представляют регрессии: различия в выходных файлах.

Выходные файлы и файлы различий генерируются в каталоге тестирования и имеют разные индексы. Например, если есть исполняемая программа или сценарий с именем `shrubbery`, а каталог тестирования называется *test1* и содержит набор входных файлов *.in*, типичный прогон тестера может выглядеть примерно так:

```
% regtest.py shrubbery test1
RegTest start.
user: mark
path: /home/mark/stuff/python/testing
time: Mon Feb 26 21:13:20 1996

FAILED: test1/t1.in (see test1/t1.in.diffs)
PASSED: test1/t2.in
FAILED: test1/t3.in (see test1/t3.in.diffs)
RegTest done: Mon Feb 26 21:13:27 1996
```

Здесь `shrubbery` прогоняется три раза, для трех заготовленных входных файлов *.in*, и результаты каждого прогона сравниваются с выдачей, сгенерированной для этих трех вариантов входных данных во время последнего проведения тестирования. Такой сценарий Python можно запускать раз в день и автоматически выявлять изменения, вызванные последними изменениями в исходном коде (возникших, например, в результате работы `cron` в Unix).

Мы уже знакомились с системными интерфейсами, используемыми в этом сценарии: большинство из них является довольно стандартными вызовами Unix, а не спецификой Python, чтобы обсуждать их. В действительности значительная часть происходящего при работе этого сценария осуществляется в программах, порождаемых вызовами `os.system`. На самом деле этот сценарий – просто *управляющая программа*, потому что он совершенно независим от тестируемой программы и читаемых им входных данных. Новые условия тестирования можно добавлять прямо на лету, помещая новые входные файлы в каталог тестирования.

С учетом того, что этот сценарий просто управляет другими программами с помощью стандартных Unix-подобных вызовов, почему здесь используется Python, а не что-нибудь другое типа C? Во-первых, равносильная программа на C была бы значительно длиннее: в ней потребовалось бы объявлять переменные, работать со структурами данных и т. д. В C все внешние сервисы существуют в единой глобальной области видимости (области компоновщика); в Python они разбиваются на пространства имен модулей (`os`, `sys`, etc.), чтобы избежать конфликтов имен. И, в отличие от C, код Python можно выполнять сразу, без компиляции и компоновки: изменения можно проверить в Python значительно быстрее. Более того, некоторая дополнительная работа позволит выполнять этот сценарий и под Windows 9x. Как вы, вероятно, уже почувствовали, Python выделяется в том, что касается переносимости и скорости программирования.

Благодаря таким преимуществам сценарии Python очень часто используются для автоматизированного тестирования. Если вас интересует использование Python для тестирования, посмотрите на веб-сайте Python (*http:// www.python.org*) другие имеющиеся инструменты (например, систему PyUnit).

Упаковка и распаковка файлов

Много-много лун тому назад (около пяти лет) я работал на машинах, где не было средств, позволявших поместить файлы в один пакет для облегчения транспортировки. Ситуация такая: есть большая группа текстовых файлов, которые нужно переместить на другой компьютер. В сегодняшнее время широко доступны средства типа `tar`,

Тест прошел неудачно?

Когда в главе 11 мы узнаем, как отправлять электронную почту из сценариев Python, вы, возможно, захотите улучшить этот сценарий так, чтобы он автоматически отправлял письмо в случае неудачи регулярно выполняемого теста. Благодаря этому не нужно будет даже помнить о необходимости проверить результаты. Конечно, можно развивать его и еще дальше.

В одной компании, в которой я работал, в сценарии тестирования компиляции были добавлены звуковые эффекты: вы слышали аплодисменты, если регрессии не было обнаружено, и совсем другие звуки в противном случае. (Советы по воспроизведению звуков смотрите в конце данной главы и в файле *playfile.py* в главе 11.)

В другой компании, где я раньше работал, ежевечерне выполнялся тестовый сценарий, который автоматически предотвращал сохранение файла исходного кода, вызвавшего регрессию в тесте, и посылал гневное письмо виновной стороне (и их начальнику). Никто не уйдет от испанской инквизиции!

позволяющие упаковать много файлов в один, который потом можно копировать, загружать на сервер, пересылать по почте или иным способом перемещать за один шаг. Даже сам Python дошел до того, что поддерживает архивы zip в стандартной библиотеке 2.0 (смотрите модуль `zipfile`).

Однако до того как мне удалось установить на своем PC такие инструменты, мне исправно служили переносимые сценарии Python. В примере 4.6 все файлы, перечисленные в командной строке, копируются в стандартный выходной поток, разделяясь маркерными строками.

Пример 4.6. PP2E\System\App\Clients\textpack.py

```
#!/usr/local/bin/python
import sys                              # загрузка системного модуля
marker = ':'*10 + 'textpak=>'           # предположительно уникальный разделитель

def pack():
    for name in sys.argv[1:]:           # для всех аргументов командной строки
        input = open(name, 'r')         # открывать очередной входной файл
        print marker + name             # запись строки-разделителя
        print input.read(),             # и запись содержимого файла

if __name__ == '__main__': pack()       # упаковка файлов, перечисленных в командной строке
```

Первая строка этого файла является комментарием Python (`#...`), но одновременно указывает путь к интерпретатору Python с помощью используемого в Unix приема для выполняемых сценариев, обсуждавшегося в главе 2. Если присвоить *textpack.py* права выполнения с помощью команды Unix *chmod*, то можно упаковывать файлы путем запуска файла этой программы прямо из оболочки Unix и переадресовывать ее стандартный вывод в файл, в котором должен находиться упакованный архив. Под Windows сценарий работает так же, но нужно вводить имя интерпретатора «python»:

```
C:\...\PP2E\System\App\Clients\test>type spam.txt
SPAM
spam

C:\......\test>python ..\textpack.py spam.txt eggs.txt ham.txt > packed.all

C:\......\test>type packed.all
::::::::::textpak=>spam.txt
```

```
SPAM
spam
::::::::::textpak=>eggs.txt
EGGS
::::::::::textpak=>ham.txt
ham
```

При таком выполнении программы создается один выходной файл с именем *packed.all*, содержащий все три входные файла и строки заголовков с именем исходного файла перед содержимым каждого файла. При таком объединении файлов в один легко осуществлять пересылку за один шаг – лишь один файл нужно копировать на гибкий диск, передавать по электронной почте и т. д. Если нужно переместить сотни файлов, получается большой выигрыш.

Однако после пересылки такого файла его нужно каким-то образом распаковать на приемном конце, чтобы воссоздать исходные файлы. Для этого нужно просмотреть архивный файл и найти строки заголовков, вставленные упаковщиком перед началом содержимого очередного файла. Это осуществляет другой простой сценарий Python, показанный в примере 4.7.

Пример 4.7. PP2E\System\App\Clients\textunpack.py

```
#!/usr/local/bin/python
import sys
from textpack import marker                  # использовать общий ключ разделителя
mlen = len(marker)                           # имена файлов после маркера

for line in sys.stdin.readlines():           # для всех входных строк
    if line[:mlen] != marker:
        print line,                          # запись действительных строк
    else:
        sys.stdout = open(line[mlen:-1], 'w')  # или создать новый выходной файл
```

Можно было бы поместить код в функцию, как в `textpack`, но в данном случае в этом мало смысла – в таком виде сценарий зависит от стандартных потоков, а не от параметров функции. Выполните его в том каталоге, куда вам нужно распаковать файлы, направив файл упакованного архива на стандартный входной поток сценария в командной строке:

```
C:\......\test\unpack>python ..\..\textunpack.py < ..\packed.all

C:\......\test\unpack>ls
eggs.txt  ham.txt   spam.txt

C:\......\test\unpack>type spam.txt
SPAM
Spam
```

Упаковка файлов «++»

Пока все хорошо: сценарии `textpack` и `textunpack` позволяют легко перемещать много файлов, не требуя много ручной работы. Но поработав некоторое время с этими и подобными им файлами, я стал замечать в них общность, почти взывавшую о повторном использовании кода. Например, почти в каждом написанном мной инструменте оболочки требовалось просматривать аргументы командной строки, переадресовывать потоки для различных источников и т. д. Кроме того, почти каждая утилита командной строки приводила к разным схемам опций командной строки, потому что каждая из них писалась с нуля.

Одним из решений таких проблем являются следующие три класса. Они определяют *иерархию классов*, предназначенную для повторного использования часто встречающегося кода инструментов оболочки. Более того, из-за повторного использования кода все программы, основанные на этой иерархии, сходны между собой в отношении опций командной строки, использования переменных окружения и другого. Как и обычно в объектно-ориентированных системах, после понимания того, какие методы нужно перегружать, такая структура классов обеспечивает экономию труда и единообразие. Модуль в примере 4.8 адаптирует логику сценария `textpack` для интеграции в эту иерархию.

Пример 4.8. PP2E\System\App\Clients\packapp.py

```
#!/usr/local/bin/python
########################################################################
# упаковать текстовые файлы в один, разделив строкой-маркером;
# % packapp.py -v -o target src src...
# % packapp.py *.txt -o packed1
# >>> apptools.appRun('packapp.py', args...)
# >>> apptools.appCall(PackApp, args...)
########################################################################
from textpack import marker
from PP2E.System.App.Kinds.redirect import StreamApp
class PackApp(StreamApp):
    def start(self):
        StreamApp.start(self)
        if not self.args:
            self.exit('packapp.py [-o target]? src src...')
    def run(self):
        for name in self.restargs():
            try:
                self.message('packing: ' + name)
                self.pack_file(name)
            except:
                self.exit('error processing: ' + name)
    def pack_file(self, name):
        self.setInput(name)
        self.write(marker + name + '\n')
        while 1:
            line = self.readline()
            if not line: break
            self.write(line)
if __name__ == '__main__':  PackApp().main()
```

Здесь `PackApp` наследует у класса `StreamApp`, импортируемого из другого модуля Python (приведенного в примере 4.10), члены и методы, обеспечивающие:

- сервисы операционной системы
- обработку командной строки
- переадресацию входных/выходных потоков

`StreamApp` обеспечивает интерфейс «чтения/записи» для переадресуемых потоков и предоставляет стандартный протокол выполнения сценариев «запуск/прогон/останов». `PackApp` просто переопределяет методы `start` и `run` для собственных целей и производит чтение и запись *себя* для доступа к своим стандартным потокам. Большинство системных интерфейсов низкого уровня скрыто в классе `StreamApp`; на языке ООП это называется *инкапсуляцией*.

Этот модуль можно выполнять как программу или импортировать в клиенте (помните, что Python устанавливает имя модуля в `__main__` при непосредственном его запуске, чтобы можно было отличить один режим от другого). При выполнении в качестве программы последняя строка создает экземпляр класса `PackApp` и запускает его, вызывая метод `main` – метод, экспортируемый `StreamApp` для запуска выполнения программы:

```
C:\......\test>python ..\packapp.py -v -o packedapp.all spam.txt eggs.txt ham.txt
PackApp start.
packing: spam.txt
packing: eggs.txt
packing: ham.txt
PackApp done.

C:\......\test>type packedapp.all
::::::::::textpak=>spam.txt
SPAM
spam
::::::::::textpak=>eggs.txt
EGGS
::::::::::textpak=>ham.txt
ham
```

Результат такой же, как при выполнении сценария *textpack.py*, но опции командной строки (`-v` – для режима вывода текстовых сообщений, `-o` – для имени выходного файла) наследуются у суперкласса `StreamApp`. Распаковщик из примера 4.9 при переносе в ОО среду выглядит аналогично, поскольку самому понятию запуска программы придана стандартная структура.

Пример 4.9. PP2E\System\App\Clients\unpackapp.py

```
#!/usr/bin/python
##########################################
# распаковать выходной файл packapp.py;
# % unpackapp.py -i packed1 -v
# apptools.appRun('unpackapp.py', args...)
# apptools.appCall(UnpackApp, args...)
##########################################
import string
from textpack import marker
from PP2E.System.App.Kinds.redirect import StreamApp

class UnpackApp(StreamApp):
    def start(self):
        StreamApp.start(self)
        self.endargs()                  # игнорировать следующие -o и т. д.
    def run(self):
        mlen = len(marker)
        while 1:
            line = self.readline()
            if not line:
                break
            elif line[:mlen] != marker:
                self.write(line)
            else:
                name = string.strip(line[mlen:])
                self.message('creating: ' + name)
                self.setOutput(name)

if __name__ == '__main__':  UnpackApp().main()
```

В этом подклассе переопределены методы start и run так, чтобы выполнять то, что требуется от этого сценария – подготовку и выполнение операции распаковки файла. Всеми деталями анализа аргументов командной строки и переадресации стандартных потоков занимается суперкласс.

```
C:\......\test\unpackapp>python ..\..\unpackapp.py -v -i ..\packedapp.all
UnpackApp start.
creating: spam.txt
creating: eggs.txt
creating: ham.txt
UnpackApp done.
C:\......\test\unpackapp>ls
eggs.txt  ham.txt   spam.txt
C:\......\test\unpackapp>type spam.txt
SPAM
spam
```

Этот сценарий выполняет ту же работу, что и первоначальный *textunpack.py*, однако мы даром получаем флаги командной строки (`-i` задает входные файлы). В действительности способов запуска классов в этой иерархии больше, чем места, которое я мог бы этому уделить здесь. Например, пара флагов командной строки `-i` заставляет сценарий читать входные данные со `stdin`, как если бы они передавались по конвейеру или переадресовывались оболочкой:

```
C:\......\test\unpackapp>type ..\packedapp.all | python ..\..\unpackapp.py -i -
creating: spam.txt
creating: eggs.txt
creating: ham.txt
```

Суперклассы иерархии приложения

В этом разделе приведен код `StreamApp` и `App` – классов, которые выполняют всю эту дополнительную работу в пользу `PackApp` и `UnpackApp`. Места для подробного разбора кода нет, поэтому для получения дополнительных сведений изучите эти листинги самостоятельно. Все это простой код Python.

Следует также отметить, что в этом разделе приведены лишь классы, используемые объектно-ориентированными разновидностями сценариев `textpack` и `textunpack`. Они представляют лишь одну ветвь всего дерева классов структуры приложения, которое можно изучить с помощью прилагаемого к книге CD (просмотрите каталог *PP2E\System\App*). Другие классы дерева предоставляют управляющие меню, внутренние файловые потоки, основанные на строках, и т. д. Вы найдете также других клиентов этой иерархии, которые делают такие вещи, как запуск других инструментов оболочки и просмотр файлов ящиков электронной почты в формате Unix.

StreamApp: добавление переадресации потоков

`StreamApp` добавляет несколько аргументов командной строки (`-i`, `-o`) и переадресацию входных/выходных потоков в более общий корневой класс `App`, который будет показан позже; `App`, в свою очередь, определяет самые общие типы поведения программы, которые в примерах 4.8, 4.9 и 4.10 наследуются всеми классами, производными от `App`.

Пример 4.10. PP2E\System\App\Kinds\redirect.py

```
########################################################################
# Подклассы App для переадресации стандартных потоков в файлы
########################################################################
```

```
import sys
from PP2E.System.App.Bases.app import App
#############################################################################
# приложение с переадресацией входных/выходных потоков
#############################################################################
class StreamApp(App):
    def __init__(self, ifile='-', ofile='-'):
        App.__init__(self)                              # инициализация суперкласса
       self.setInput( ifile or self.name + '.in')     # имена файлов ввода/вывода по умолчанию,
        self.setOutput(ofile or self.name + '.out')    # если нет аргументов '-i', '-o'
    def closeApp(self):                                 # не __del__
        try:
            if self.input != sys.stdin:                 # возможна переадресация,
                self.input.close()                      # если еще открыт
        except: pass
        try:
            if self.output != sys.stdout:               # не закрывайте stdout!
                self.output.close()                     # input/output существуют?
        except: pass
    def help(self):
        App.help(self)
        print '-i <input-file |"-">  (default: stdin  or per app)'
        print '-o <output-file|"-">  (default: stdout or per app)'
    def setInput(self, default=None):
        file = self.getarg('-i') or default or '-'
        if file == '-':
            self.input = sys.stdin
            self.input_name = '<stdin>'
        else:
            self.input = open(file, 'r')                # cmdarg | funcarg | stdin
            self.input_name = file                      # cmdarg '-i -' тоже действует
    def setOutput(self, default=None):
        file = self.getarg('-o') or default or '-'
        if file == '-':
            self.output = sys.stdout
            self.output_name = '<stdout>'
        else:
            self.output = open(file, 'w')               # перехвачена ошибка в main()
            self.output_name = file                     # сделать резервные копии?
class RedirectApp(StreamApp):
    def __init__(self, ifile=None, ofile=None):
        StreamApp.__init__(self, ifile, ofile)
        self.streams = sys.stdin, sys.stdout
        sys.stdin    = self.input                   # для raw_input, stdin
        sys.stdout   = self.output                  # для print, stdout
    def closeApp(self):                             # не __del__
        StreamApp.closeApp(self)                    # закрыть файлы?
        sys.stdin, sys.stdout = self.streams        # сбросить файлы sys
#############################################################################
# добавить как подмениваемый (или использовать множественное наследование...)
#############################################################################
class RedirectAnyApp:
    def __init__(self, superclass, *args):
        apply(superclass.__init__, (self,) + args)
        self.super   = superclass
        self.streams = sys.stdin, sys.stdout
        sys.stdin    = self.input                   # для raw_input, stdin
```

```
        sys.stdout   = self.output                  # для print, stdout
    def closeApp(self):
        self.super.closeApp(self)                   # правильный поступок
        sys.stdin, sys.stdout = self.streams        # сброс файлов sys
```

App: корневой класс

Вершина иерархии понимает, что значит быть приложением оболочки, но не знает, как решать задачу конкретной утилиты (эти части реализуют подклассы). `App`**, приведенный в примере 4.11, экспортирует часто используемые инструменты в стандартном и упрощенном интерфейсе и настраиваемый протокол методов** `start/run/stop`**, служащий абстракцией выполнения сценария. Он также превращает объекты приложения в объекты, похожие на файловые: например, когда приложение читает само себя, оно в действительности читает тот источник, который соединен с его стандартным входом другим суперклассом дерева (как** `StreamApp`**).**

Пример 4.11. PP2E\System\App\Bases\app.py

```
################################################################################
# иерархия классов приложения для работы с компонентами верхнего уровня;
# App является корневым классом иерархии App, расширяемым в других файлах;
################################################################################

import sys, os, traceback
AppError = 'App class error'                        # ошибки, которые возникли в этом классе

class App:                                          # корневой класс
    def __init__(self, name=None):
        self.name    = name or self.__class__.__name__    # самый нижний класс
        self.args    = sys.argv[1:]
        self.env     = os.environ
        self.verbose = self.getopt('-v') or self.getenv('VERBOSE')
        self.input   = sys.stdin
        self.output  = sys.stdout
        self.error   = sys.stderr                   # stdout может быть переадресован
    def closeApp(self):                             # не __del__: могут быть ссылки?
        pass                                        # на этом уровне ничего
    def help(self):
        print self.name, 'command-line arguments:'  # расширить в подклассе
        print '-v (verbose)'

    ####################################
    # сервисы окружения сценария
    ####################################

    def getopt(self, tag):
        try:                                        # проверка аргумента "-x"
            self.args.remove(tag)                   # ненастоящий argv: > 1 App?
            return 1
        except:
            return 0
    def getarg(self, tag, default=None):
        try:                                        # получить аргумент "-x val"
            pos = self.args.index(tag)
            val = self.args[pos+1]
            self.args[pos:pos+2] = []
            return val
        except:
           return default                   # None: отсутствует, значения по умолчанию нет
```

```
def getenv(self, name, default=''):
    try:                                            # получит переменную окружения "$x"
        return self.env[name]
    except KeyError:
        return default
def endargs(self):
    if self.args:
        self.message('extra arguments ignored: ' + `self.args`)
        self.args = []
def restargs(self):
    res, self.args = self.args, []                  # аргументов/опций больше нет
    return res
def message(self, text):
    self.error.write(text + '\n')                   # stdout может быть переадресован
def exception(self):
    return (sys.exc_type, sys.exc_value)            # последняя исключительная ситуация
def exit(self, message='', status=1):
    if message:
        self.message(message)
    sys.exit(status)
def shell(self, command, fork=0, inp=''):
    if self.verbose:
        self.message(command)                       # как насчет ipc?
    if not fork:
        os.system(command)                          # выполнить команду оболочки
    elif fork == 1:
        return os.popen(command, 'r').read()        # получить ее вывод
    else:                                           # readlines тоже?
        pipe = os.popen(command, 'w')
        pipe.write(inp)                             # послать туда ввод
        pipe.close()

#############################################################################################
# методы потоков input/output для самого приложения; переопределить в подклассах,
# если не используются файлы, или назначить self.input/output объектам типа файл;
#############################################################################################

def read(self, *size):
    return apply(self.input.read, size)
def readline(self):
    return self.input.readline()
def readlines(self):
    return self.input.readlines()
def write(self, text):
    self.output.write(text)
def writelines(self, text):
    self.output.writelines(text)

#############################################################################
# для запуска приложения main() является протоколом исполнения start/run/stop;
#############################################################################

def main(self):
    res = None
    try:
        self.start()
        self.run()
        res = self.stop()                   # необязательный возврат значения
    except SystemExit:                      # игнорировать, если от exit()
```

```
            pass
        except:
            self.message('uncaught: ' + `self.exception()`)
            traceback.print_exc()
        self.closeApp()
        return res

    def start(self):
        if self.verbose: self.message(self.name + ' start.')
    def stop(self):
        if self.verbose: self.message(self.name + ' done.')
    def run(self):
        raise AppError, 'run must be redefined!'
```

Зачем здесь используются классы?

Теперь, когда я привел весь этот код, некоторые читатели могут задать естественный вопрос: «И зачем все эти проблемы?» С учетом объема *дополнительного* кода в ОО-версиях этих сценариев такой вопрос будет совершенно законным. Большая часть кода, приведенного в примере 4.11, является логикой общего назначения, которая может использоваться многими приложениями. Но это не объясняет, почему объектно-ориентированные сценарии `packapp` и `unpackapp` больше по размеру, чем первоначальные эквивалентные `textpack` и `textunpack`, не использующе ООП.

Ответ станет более очевидным после нескольких случаев, когда вам *не нужно* будет писать код, чтобы достичь цели, но есть несколько конкретных преимуществ, которые можно суммировать в данном месте:

Инкапсуляция

Клиентам `StreamApp` не нужно помнить все системные интерфейсы Python, потому что `StreamApp` экспортирует собственное унифицированное представление. Например, аргументы, потоки и переменные окружения разбросаны по нескольким модулям Python (например, `sys.argv`, `sys.stdout`, `os.environ`); в данных классах они собраны в едином месте.

Стандартизация

С точки зрения пользователя оболочки, у клиентов `StreamApp` одинаковое внешнее представление, потому что они наследуют одни и те же интерфейсы к внешнему миру у своих суперклассов (например, флаги `-i` и `-v`).

Сопровождение

Отладка всего общего кода в суперклассах `App` и `StreamApp` происходит только один раз. Кроме того, сосредоточение кода в суперклассах облегчает его понимание и возможные изменения в будущем.

Повторное использование

Такая структура предоставляет дополнительные утилиты с готовым кодом, которые иначе пришлось бы заново кодировать в каждом создаваемом сценарии (например, извлечение аргументов командной строки). Это касается как настоящего, так и будущего – сервисы, добавляемые в корневой класс `App`, сразу можно использовать и модифицировать во всех приложениях, производных от этой иерархии.

Практичность

Поскольку доступ к файлам в `PackApp` и `UnpackApp` не кодируется жестко, они могут легко усваивать новое поведение в результате изменения класса, которому наследуют. Если задать соответствующий суперкласс, `PackApp` и `UnpackApp` могут с такой

же легкостью читать и писать в строки и сокеты, как в текстовые файлы и стандартные потоки.

Повторное использование кода является, вероятно, самым большим преимуществом программ, написанных с использованием классов, хотя это может не быть очевидным, пока вы не начнете писать крупные системы, основанные на классах. Например, в главе 9 «Более крупные примеры GUI» мы будем повторно использовать объектно-ориентированные сценарии упаковщика и распаковщика, вызывая их из графического интерфейса меню следующим образом:

```
from PP2E.System.App.Clients.packapp import PackApp
... получить в диалоге входные данные, шаблон имени файла для поиска
app = PackApp(ofile=output)       # выполнение с переадресацией вывода
app.args = filenames              # переустановить список аргументов командной строки
app.main()

from PP2E.System.App.Clients.unpackapp import UnpackApp
... получить в диалоге входные данные
app = UnpackApp(ifile=input)          # выполнить с данными из входного файла
app.main()                            # выполнить класс приложения
```

Поскольку эти классы инкапсулируют понятие потока, их можно импортировать и вызывать, а не только выполнять как сценарии верхнего уровня. Кроме того, есть два пути для повторного использования их кода: они не только экспортируют общие системные интерфейсы для повторного использования в подклассах, но могут использоваться в качестве программных *компонентов*, как в предыдущем листинге. Полный исходный код этих клиентов можно найти в каталоге *PP2E\Gui\Shellgui*.

Конечно, Python не навязывает использование ООП, и многие вещи можно делать с помощью более простых функций и сценариев. Но научившись структурировать деревья классов для повторного использования, стоит тратить усилия на ООП, которые в конечном итоге обычно окупаются.

Дружественные пользователю средства для запуска программ

Предположим, что требуется поставлять программы Python клиентам, находящимся на ранних стадиях перехода от пользователя компьютера к программисту. Возможно, вы поставляете приложение Python пользователям, не имеющим технического образования, или хотите поместить ряд ярких демонстрационных программ на CD-ROM в книге по Python. Какими бы ни были причины, от некоторых пользователей ваших программ нельзя ожидать большего, чем способности щелкнуть мышкой, и уж никак не умения отредактировать файлы конфигурации системы, чтобы настроить PATH и PYTHONPATH в соответствии с предположениями вашей программы. Вашим программам придется произвести настройку самостоятельно.

К счастью, сценарии Python могут и это делать. В следующих двух разделах мы рассмотрим два модуля, задачей которых является автоматический запуск программ с минимальными допущениями относительно конфигурации окружения:

- *Launcher.py* служит библиотекой для автоматического конфигурирования окружения оболочки, подготавливая ее к запуску сценария Python. Ее можно использовать для настройки необходимых переменных оболочки – как системного пути поиска программ PATH (используемого при поиске исполняемого модуля «python»), так и пути поиска модулей PYTHONPATH (используемого для поиска импортируемых модулей внутри сценария). Поскольку такие настройки перемен-

ных, сделанные в родительской программе, *наследуются* порожденными дочерними программами, этот интерфейс позволяет сценариям осуществлять предварительную настройку путей поиска для других сценариев.

- *LaunchBrowser.py* предназначен для переносимого обнаружения и запуска программы броузера Интернета, чтобы просмотреть локальный файл или удаленную веб-страницу. Он использует средства, имеющиеся в *Launcher.py* для поиска броузера, который можно запустить.

Оба эти модуля разработаны так, что могут повторно использоваться в любом контексте, где от программного обеспечения требуется дружественное отношение к пользователю. Благодаря автоматическому поиску файлов и настройке окружения пользователи могут избежать (хотя бы временно) необходимости изучать сложности настройки окружения.

Клиенты модуля Launcher

Эти два модуля из данного раздела действуют во многих примерах этой книги. В действительности мы уже использовали некоторые из этих инструментов. Сценарий `launchmodes`, с которым мы познакомились в конце предыдущей главы, импортировал функции `Launcher`, чтобы найти путь к локальному интерпретатору *python.exe*, который требуется в вызовах `os.spawnv`. В этом сценарии можно было бы предположить, что тот, кто устанавливает его на своей машине, отредактирует его исходный код, добавив реальное местонахождение Python; однако большинство потенциальных пользователей отделено от необходимых даже для такой задачи технических навыков тысячами световых лет.[1] Гораздо лучше пожертвовать пренебрежимо малым временем при начальном запуске для автоматического обнаружения Python.

Два модуля, приведенные в примерах 4.14 и 4.15, вместе с `launchmodes` образуют также ядро *программ запуска демонстрационных примеров*, находящихся в вершине дистрибутива примеров на CD этой книги. Ничто не может заменить возможность собственными глазами посмотреть на программы в действии, поэтому я хотел сделать запуск примеров Python из книги как можно более легким. В идеале они должны были бы запускаться прямо с CD с помощью щелчка и не требовать от читателя прохождения сложной процедуры настройки окружения.

Однако для многих демонстрационных примеров требуется импорт из других каталогов, поэтому каталоги с пакетами модулей из книги должны быть прописаны в PYTHONPATH; простого щелчка по значку для некоторых программ недостаточно. Более того, в самом начале нельзя ожидать и того, что пользователи добавили исполняемый файл интерпретатора Python в системный путь поиска; название «python» может ничего не говорить оболочке.

По крайней мере на тех платформах, которые проверены на сегодняшний день, следующие модули решают такие проблемы настройки. Например, сценарий *Launch_PyDemos.pyw* в корневом каталоге автоматически настраивает систему и окружение выполнения Python с помощью средств *Launcher.py*, а затем порождает *PyDemos.py*, графический интерфейс демонстраций на Tkinter, с которым мы познакомимся далее в этой книге. PyDemos, в свою очередь, запускает другие программы с

[1] Вам, гуру и волшебникам, придется поверить мне на слово. Первое, что вы узнаете, разъезжая по всему свету и преподавая Python, это то, что разработчики слишком многое считают само собой разумеющимся. Например, в книге «Learning Python» мы с моим соавтором предлагали читателям делать такие вещи, как «открыть файл в своем любимом текстовом редакторе» и «запустить командную консоль DOS». Не было недостатка в электронной почте от начинающих с вопросами о том, что же такое мы имели в виду.

помощью launchmodes, которые наследуют настройки окружения, сделанные вначале. В итоге щелчок по любому из сценариев Launch_* запускает программы Python, даже если вы не прикоснулись к настройкам окружения.

Конечно, по-прежнему требуется установить Python, если это не сделано раньше, но самоустанавливающаяся программа Python для Windows тоже требует лишь простого щелчка по ней. Поскольку поиск и настройка требуют дополнительного времени, к вашей же выгоде будет в конечном счете сделать необходимую настройку окружения и выполнять программы типа PyDemos напрямую, а не посредством запускающих сценариев. Можно многое сказать в пользу мгновенного удовлетворения, если это относится к программному обеспечению.

Далее в этой книге эти инструменты можно будет увидеть и в другом контексте. Например, интерфейс электронной почты PyMail, с которым мы познакомимся в главе 11, с помощью Launcher находит собственный файл исходного кода. Поскольку невозможно предвидеть, из какого каталога он будет запущен, все, что можно сделать в этом случае, – это просто поискать его. В другом примере GUI, big_gui, аналогичное средство Launcher будет использовано для нахождения готовых демонстрационных программ из дистрибутива исходного кода Python в произвольных, заранее неизвестных каталогах компьютера.

Сценарий LaunchBrowser в примере 4.15 также использует Launcher для нахождения подходящих веб-броузеров, и сам используется для запуска демонстрационных программ Интернета в графических интерфейсах запуска PyDemos и PyGadgets, то есть Launcher запускает PyDemos, который запускает LaunchBrowser, который использует Launcher. Оптимизируя общность, эти модули также оптимизируют многократное использование.

Запуск программ без настроек окружения

Поскольку файл *Launcher.py* хорошо документирован, я обойду в своем повествовании его тонкие места. Вместо этого я лишь отмечу, что все его функции полезны сами по себе, но главной точкой входа служит расположенная ближе к концу функция launchBookExamples; чтобы получить общую картину, нужно продвигаться вверх от конца файла.

Функция launchBookExamples использует все остальные, чтобы настроить окружение, а затем запустить одну или несколько программ для выполнения в этом окружении. В действительности сценарии верхнего уровня для запуска демонстрационных программ, показанные в примерах 4.12 и 4.13, всего лишь просят эту функцию запустить демонстрационные программы GUI, с которыми мы встретимся позже (например, *PyDemos.pyw*, *PyGadgets_bar.pyw*). Поскольку GUI порождаются косвенно через этот интерфейс, все порождаемые ими программы наследуют настройки окружения.

Пример 4.12. PP2E\Launch_PyDemos.pyw

```
#!/bin/env python
#########################################################################
# PyDemos + первоначальные поиск/настройка окружения запускайте, если еще
# не настроили свои пути, однако сначала нужно все-таки установить Python
#########################################################################

import Launcher
Launcher.launchBookExamples(['PyDemos.pyw'], 0)
```

Пример 4.13. PP2E\Launch_PyGadgets_bar.pyw

```
#!/bin/env python
```

```
##############################################################################
# PyGadgets_bar + первоначальные поиск/настройка окружения запускайте, если еще
# не настроили свои пути, однако сначала нужно все-таки установить Python
##############################################################################

import Launcher
Launcher.launchBookExamples(['PyGadgets_bar.pyw'], 0)
```

При непосредственном запуске *PyDemos.pyw* и *PyGadgets_bar.pyw* используют настройку конфигурации соответствующей машины. Иными словами, `Launcher` **фактически *скрывает* детали конфигурации от GUI, помещая их в слой конфигурирующей программы. Чтобы понять, как это происходит, изучите пример 4.14.**

Пример 4.14. PP2E\Launcher.py

```
#!/usr/bin/env python
"""
------------------------------------------------------------------------
Средства, позволяющие находить файлы и выполнять демонстрационные программы Python, даже когда
окружение еще не сконфигурировано вручную. Например, если Python уже установлен, можно
запускать демонстрационные программы Tk прямо с CD книги двойным щелчком по значку этого файла,
не редактируя предварительно файлов конфигурации окружения. Предполагается, что сначала
установлен Python (сделайте двойной щелчок по самоустанавливающемуся exe-файлу python на CD), и
делается попытка найти Python и дистрибутив примеров на машине. Перед запуском сценариев
настраиваются системный путь поиска и путь поиска модулей Python: это возможно только потому,
что порождаемые программы наследуют настройки окружения как в Windows, так и в Linux. Для
повышения скорости может потребоваться отредактировать список каталогов, в которых
производится поиск, а также запустить один из файлов Config/setup-pp при начальном запуске,
чтобы избежать такого поиска. Этот сценарий хорошо воспринимает уже настроенные значения путей
и служит для демонстрации такой обработки каталогов, которая не зависит от платформы. Программы
Python всегда можно запустить под Windows щелчком мыши (или породив командой DOS 'start'), но
для многих примеров из книги требуется также путь поиска модулей.
------------------------------------------------------------------------
"""
import sys, os, string
def which(program, trace=1):
    """
    Искать программу во всех каталогах из переменной пути системного поиска PATH;
    вернуть полный путь к программе, если она найдена, или None. Не обрабатывает
    символические ссылки в Unix (где можно было бы также просто выполнить команду
    оболочки 'which' с помощью os.popen) и может также помочь проверить, является ли
    файл действительно исполняемым, с помощью os.stat и модуля stat, посредством
    кода такого типа : os.stat(filename)[stat.ST_MODE] & 0111
    """
    try:
        ospath = os.environ['PATH']
    except:
        ospath = '' # не страшно, если пустой
    systempath = string.split(ospath, os.pathsep)
    if trace: print 'Looking for', program, 'on', systempath
    for sysdir in systempath:
        filename = os.path.join(sysdir, program)      # вставляет os.sep
        if os.path.isfile(filename):                  # существует ли файл?
            if trace: print 'Found', filename
            return filename
        else:
            if trace: print 'Not at', filename
    if trace: print program, 'not on system path'
    return None
```

```
def findFirst(thisDir, targetFile, trace=0):
    """
    Рекусивный поиск в каталоге thisDir и ниже файла или каталога с именем
    targetFile. Подобна find.find стандартной библиотеки, но не использует шаблоны
    имени, следует по ссылкам unix и останавливается на первом файле с совпавшим
    именем. targetFile должен быть простым базовым именем, а не путем каталога.
    """
    if trace: print 'Scanning', thisDir
    for filename in os.listdir(thisDir):                  # пропустить . и ..
        if filename in [os.curdir, os.pardir]:            # если да
            continue
        elif filename == targetFile:                      # проверить совпадение имени
            return os.path.join(thisDir, targetFile)      # на этом остановиться
        else:
            pathname = os.path.join(thisDir, filename)    # войти в подкаталоги
            if os.path.isdir(pathname):                   # остановиться при первом совпадении
                below = findFirst(pathname, targetFile, trace)
                if below: return below
def guessLocation(file, isOnWindows=(sys.platform[:3]=='win'), trace=1):
    """
    Попытаться найти каталог, в который установлен файл, просматривая обычные места
    для данной платформы. Измените опробуемые пути соответственно своей машине.
    """
    cwd = os.getcwd()                           # каталог, в котором запущен ру
    tryhere = cwd + os.sep + file               # или os.path.join(cwd, file)
    if os.path.exists(tryhere):                 # не искать, если он здесь
        return tryhere                          # findFirst(cwd,file) descends
    if isOnWindows:
        tries = []
        for pydir in [r'C:\Python20', r'C:\Program Files\Python']:
            if os.path.exists(pydir):
                tries.append(pydir)
        tries = tries + [cwd, r'C:\Program Files']
        for drive in 'CGDEF':
            tries.append(drive + ':\\')
    else:
        tries = [cwd, os.environ['HOME'], '/usr/bin', '/usr/local/bin']
    for dir in tries:
        if trace: print 'Searching for %s in %s' % (file, dir)
        try:
            match = findFirst(dir, file)
        except OSError:
            if trace: print 'Error while searching', dir     # skip bad drives
        else:
            if match: return match
    if trace: print file, 'not found! - configure your environment manually'
    return None
PP2EpackageRoots = [                                # путь поиска модулей python
   #'%sPP2E' % os.sep,                              # добавьте свой собственный
    '']                                             # '' добавляет корень examplesDir
def configPythonPath(examplesDir, packageRoots=PP2EpackageRoots, trace=1):
    """
    Установка пути к каталогам для поиска модулей Python, как требуется для выполнения
    программ в дистрибутиве примеров книги, если он еще не сконфигурирован.
    Добавляет корень пакетов примеров, а также корни вложенных пакетов. Это соответствует
    настройкам файла конфигурации setup-pp*. Присвоения os.environ внутренне вызывают
    os.putenv в 1.5, поэтому данные установки наследуются порождаемыми программами.
    Автоматически производится поиск в каталоге с исходным кодом библиотеки Python
```

```
    и в '.'; для unix|win значения os.sep: '/' | '\\', os.pathsep: ':' | ';'. sys.path
    используется только в этом процессе – нужно установить os.environ.
    Новые каталоги добавляются вперед, на случай присутствия двух инсталляций.
    Можно было бы также попробовать выполнить в этом процессе setup-pp* для данной платформы,
    но это непереносимо, медленно и чревато ошибками.
    """
    try:
        ospythonpath = os.environ['PYTHONPATH']
    except:
        ospythonpath = '' # не страшно, если пучтой
    if trace: print 'PYTHONPATH start:\n', ospythonpath
    addList = []
    for root in packageRoots:
        importDir = examplesDir + root
        if importDir in sys.path:
            if trace: print 'Exists', importDir
        else:
            if trace: print 'Adding', importDir
            sys.path.append(importDir)
            addList.append(importDir)
    if addList:
        addString = string.join(addList, os.pathsep) + os.pathsep
        os.environ['PYTHONPATH'] = addString + ospythonpath
        if trace: print 'PYTHONPATH updated:\n', os.environ['PYTHONPATH']
    else:
        if trace: print 'PYTHONPATH unchanged'
def configSystemPath(pythonDir, trace=1):
    """
    Добавьте каталог, в котором находится интерпретатор python в системные пути, если хотите
    """
    try:
        ospath = os.environ['PATH']
    except:
        ospath = '' # okay if not set
    if trace: print 'PATH start', ospath
    if (string.find(ospath, pythonDir) == -1 and                  # не найден?
        string.find(ospath, string.upper(pythonDir)) == -1):      # отличие в регистре?
        os.environ['PATH'] = ospath + os.pathsep + pythonDir
        if trace: print 'PATH updated:', os.environ['PATH']
    else:
        if trace: print 'PATH unchanged'
def runCommandLine(pypath, exdir, command, isOnWindows=0, trace=1):
    """
    Запуск команды python как независимой программы/процесса на этой платформе,
    используя pypath как путь к исполняемому файлу интерпретатора Python и exdir
    как корневой каталог установленных примеров. В windows нужен полный путь
    к python, в unix - нет. В windows os.system('start ' + command) действует
    аналогично, но файлы .py вызывают окно консоли dos для ввода/вывода.
    Также можно использовать launchmodes.py, но pypath уже известен.
    """
    command = exdir + os.sep + command          # корень в дереве примеров
    os.environ['PP2E_PYTHON_FILE'] = pypath     # экспорт каталогов для
    os.environ['PP2E_EXAMPLE_DIR'] = exdir      # использования в запущенных программах
    if trace: print 'Spawning:', command
    if isOnWindows:
        os.spawnv(os.P_DETACH, pypath, ('python', command))
    else:
        cmdargs = [pypath] + string.split(command)
```

```
        if os.fork() == 0:
            os.execv(pypath, cmdargs)           # выполнение в дочернем процессе
def launchBookExamples(commandsToStart, trace=1):
    """
    Точка входа верхнего уровня: поиск exe-файла python и
    каталога примеров, настройка окружения, запуск программ
    """
    isOnWindows  = (sys.platform[:3] == 'win')
    pythonFile   = (isOnWindows and 'python.exe') or 'python'
    examplesFile = 'README-PP2E.txt'
    if trace:
        print os.getcwd(), os.curdir, os.sep, os.pathsep
        print 'starting on %s...' % sys.platform
    # найти исполняемый файл python: сначала проверка системного пути,
    # затем попробовать угадать
    pypath = which(pythonFile) or guessLocation(pythonFile, isOnWindows)
    assert pypath
    pydir, pyfile = os.path.split(pypath)                 # на 1 вверх от файла
    if trace:
        print 'Using this Python executable:', pypath
        raw_input('Press <enter> key')
    # найти корневой каталог примеров: проверить cwd и другие
    expath = guessLocation(examplesFile, isOnWindows)
    assert expath
    updir  = string.split(expath, os.sep)[:-2]          # на 2 вверх от файла
    exdir  = string.join(updir,   os.sep)               # к родительскому каталогу пакета PP2E
    if trace:
        print 'Using this examples root directory:', exdir
        raw_input('Press <enter> key')
    # если нужно, экспорт путей python и системного
    configSystemPath(pydir)
    configPythonPath(exdir)
    if trace:
        print 'Environment configured'
        raw_input('Press <enter> key')
    # запуск программ
    for command in commandsToStart:
        runCommandLine(pypath, os.path.dirname(expath), command, isOnWindows)

if __name__ == '__main__':
    #
    # если нет аргументов, запустить все нижеследующие программы, либо
    # остальные аргументы командной строки дают одну команду для запуска
    #
    if len(sys.argv) == 1:
        commandsToStart = [
            'Gui/TextEditor/textEditor.pyw',          # годится любой слэш
            'Lang/Calculator/calculator.py',          # os нормализует путь
            'PyDemos.pyw',
           #'PyGadgets.py',
            'echoEnvironment.pyw'
        ]
    else:
        commandsToStart = [ string.join(sys.argv[1:], ' ') ]
    launchBookExamples(commandsToStart)
    import time
    if sys.platform[:3] == 'win': time.sleep(10)      # чтобы прочитать сообщения
                                                      # после запуска мышью
```

Один из способов разобраться со сценарием Launcher – проследить за всеми выводимыми им попутно сообщениями. При самостоятельном выполнении и не установленной PYTHONPATH этот сценарий находит подходящий интерпретатор Python и корневой каталог примеров (разыскивая находящийся в нем файл README), использует полученные результаты для настройки PATH и PYTHONPATH при необходимости и запускает примеры программ из имеющегося в коде списка. Вот для иллюстрации результат запуска под Windows с пустой PYTHONPATH:

```
C:\temp\examples>set PYTHONPATH=
C:\temp\examples>python Launcher.py
C:\temp\examples . \ ;
starting on win32...
Looking for python.exe on ['C:\\WINDOWS', 'C:\\WINDOWS',
'C:\\WINDOWS\\COMMAND', 'C:\\STUFF\\BIN.MKS', 'C:\\PROGRAM FILES\\PYTHON']
Not at C:\WINDOWS\python.exe
Not at C:\WINDOWS\python.exe
Not at C:\WINDOWS\COMMAND\python.exe
Not at C:\STUFF\BIN.MKS\python.exe
Found C:\PROGRAM FILES\PYTHON\python.exe
Using this Python executable: C:\PROGRAM FILES\PYTHON\python.exe
Press <enter> key
Using this examples root directory: C:\temp\examples
Press <enter> key
PATH start C:\WINDOWS;C:\WINDOWS;C:\WINDOWS\COMMAND;C:\STUFF\BIN.MKS;
C:\PROGRAM FILES\PYTHON
PATH unchanged
PYTHONPATH start:
Adding C:\temp\examples\Part3
Adding C:\temp\examples\Part2
Adding C:\temp\examples\Part2\Gui
Adding C:\temp\examples
PYTHONPATH updated:
C:\temp\examples\Part3;C:\temp\examples\Part2;C:\temp\examples\Part2\Gui;
C:\temp\examples;
Environment configured
Press <enter> key
Spawning: C:\temp\examples\Part2/Gui/TextEditor/textEditor.pyw
Spawning: C:\temp\examples\Part2/Lang/Calculator/calculator.py
Spawning: C:\temp\examples\PyDemos.pyw
Spawning: C:\temp\examples\echoEnvironment.pyw
```

Четыре программы запускаются с предварительной настройкой PATH и PYTHONPATH в соответствии с расположением программы-интерпретатора Python, адресом каталога дистрибутива примеров и списком требуемых элементов PYTHONPATH в переменной сценария PP2EpackageRoots.

Каталоги PYTHONPATH, добавляемые на этапе предварительной настройки, могут оказаться другими во время выполнения этого сценария, потому что переменная PP2EpackageRoots может получить совсем другое значение к тому моменту, когда будет записываться CD для этой книги. В действительности, чтобы сделать этот пример более интересным, приведенные листинги были получены в то время, когда требования к PYTHONPATH книги были значительно более сложными, чем они стали теперь:

```
PP2EpackageRoots = [
    '%sPart3' % os.sep,      # путь поиска модулей python,
```

```
'%sPart2' % os.sep,       # требуемый демо-программами книги
'%sPart2%sGui' % ((os.sep,)*2),
'']                       # '' добавляет каталог examplesDir
```

С тех пор дерево было реорганизовано так, чтобы к пути поиска модулей нужно было добавить только один каталог – тот, в котором содержится корневой каталог *PP2E*. Благодаря этому облегчается настройка (теперь в PYTHONPATH добавляется только один элемент), но для общности код по-прежнему поддерживает список элементов. Подобно большинству разработчиков, я не могу удержаться, чтобы не поработать с каталогами.

При использовании запускающим сценарием PyDemos `Launcher` не останавливается в процессе исполнения в ожидании нажатия клавиш (аргумент `trace` передается равным `false`). Вот вывод, получаемый при использовании модуля для запуска PyDemos с PYTHONPATH уже установленным таким образом, что он содержит все необходимые каталоги; сценарий избегает включения избыточных установок и сохраняет те, которые уже есть в вашем окружении:

```
C:\PP2ndEd\examples>python Launch_PyDemos.pyw
Looking for python.exe on ['C:\\WINDOWS', 'C:\\WINDOWS',
'C:\\WINDOWS\\COMMAND', 'C:\\STUFF\\BIN.MKS', 'C:\\PROGRAM FILES\\PYTHON']
Not at C:\WINDOWS\python.exe
Not at C:\WINDOWS\python.exe
Not at C:\WINDOWS\COMMAND\python.exe
Not at C:\STUFF\BIN.MKS\python.exe
Found C:\PROGRAM FILES\PYTHON\python.exe
PATH start C:\WINDOWS;C:\WINDOWS;C:\WINDOWS\COMMAND;C:\STUFF\BIN.MKS;
C:\PROGRAM FILES\PYTHON
PATH unchanged
PYTHONPATH start:
C:\PP2ndEd\examples\Part3;C:\PP2ndEd\examples\Part2;C:\PP2ndEd\examples\
Part2\Gui;C:\PP2ndEd\examples
Exists C:\PP2ndEd\examples\Part3
Exists C:\PP2ndEd\examples\Part2
Exists C:\PP2ndEd\examples\Part2\Gui
Exists C:\PP2ndEd\examples
PYTHONPATH unchanged
Spawning: C:\PP2ndEd\examples\PyDemos.pyw
```

И наконец, вот вывод запуска с включенной трассировкой на моей системе Linux; поскольку `Launcher` написан с переносимым кодом Python и библиотечными вызовами, конфигурирование окружения и поиск каталогов нормально работают и там:

```
[mark@toy ~/PP2ndEd/examples]$ unsetenv PYTHONPATH
[mark@toy ~/PP2ndEd/examples]$ python Launcher.py
/home/mark/PP2ndEd/examples . / :
starting on linux2...
Looking for python on ['/home/mark/bin', '.', '/usr/bin', '/usr/bin', '/usr/local/ bin', '/usr/
X11R6/bin', '/bin', '/usr/X11R6/bin', '/home/mark/
bin', '/usr/X11R6/bin', '/home/mark/bin', '/usr/X11R6/bin']
Not at /home/mark/bin/python
Not at ./python
Found /usr/bin/python
Using this Python executable: /usr/bin/python
Press <enter> key
Using this examples root directory: /home/mark/PP2ndEd/examples
```

```
Press <enter> key
PATH start /home/mark/bin:.:/usr/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/bin:/ usr
/X11R6/bin:/home/mark/bin:/usr/X11R6/bin:/home/mark/bin:/usr/X11R6/bin
PATH unchanged
PYTHONPATH start:
Adding /home/mark/PP2ndEd/examples/Part3
Adding /home/mark/PP2ndEd/examples/Part2
Adding /home/mark/PP2ndEd/examples/Part2/Gui
Adding /home/mark/PP2ndEd/examples
PYTHONPATH updated:
/home/mark/PP2ndEd/examples/Part3:/home/mark/PP2ndEd/examples/Part2:/home/
mark/PP2ndEd/examples/Part2/Gui:/home/mark/PP2ndEd/examples:
Environment configured
Press <enter> key
Spawning: /home/mark/PP2ndEd/examples/Part2/Gui/TextEditor/textEditor.py
Spawning: /home/mark/PP2ndEd/examples/Part2/Lang/Calculator/calculator.py
Spawning: /home/mark/PP2ndEd/examples/PyDemos.pyw
Spawning: /home/mark/PP2ndEd/examples/echoEnvironment.pyw
```

Во всех этих запусках интерпретатор Python был обнаружен в системном пути поиска, поэтому действительного поиска не производилось (строки «Not at» недалеко от начала представляют функцию `which` модуля). Чуть позднее мы воспользуемся функциями `which` и `guessLocation` модуля `Launcher` для нахождения веб-броузеров с запуском поиска в деревьях каталогов стандартной установки. В последствии в этой книге данный модуль будет использоваться иными способами, например, для поиска демонстрационных программ и файлов с исходным кодом, находящихся на машине, с помощью вызовов вида:

```
C:\temp>python
>>> from PP2E.Launcher import guessLocation
>>> guessLocation('hanoi.py')
Searching for hanoi.py in C:\Program Files\Python
Searching for hanoi.py in C:\temp\examples
Searching for hanoi.py in C:\Program Files
Searching for hanoi.py in C:\
'C:\\PP2ndEd\\cdrom\\Python1.5.2\\SourceDistribution\\Unpacked\\Python-1.5.2
\\Demo\\tkinter\\guido\\hanoi.py'
>>> from PP2E.Launcher import findFirst
>>> findFirst('.', 'PyMailGui.py')
'.\\examples\\Internet\\Email\\PyMailGui.py'
```

В таком поиске нет необходимости, если можно получить из переменной окружения хотя бы часть пути к файлу; например, сценарии путей внутри дерева примеров *PP2E* можно назвать, присоединив к переменной окружения `PP2EHOME` оставшуюся часть пути сценария (предполагая, что оставшаяся часть пути сценария не меняется и можно положиться на то, что эта переменная окружения установлена всюду).

Некоторые сценарии могут также образовывать относительные пути к другим сценариям с помощью указателя исходного каталога `sys.path[0]`, добавляемого к путям для импорта (смотрите раздел «Текущий рабочий каталог» главы 2). Но если файл может оказаться в произвольном месте, лучшим решением для сценариев оказывается поиск типа показанного выше. К примеру, предшествующая программа *hanoi.py* может находиться в любом месте на машине (или вообще отсутствовать); поиск является более дружественной пользователю альтернативой, чем отказ от борьбы.

Переносимый запуск веб-броузеров

Современные веб-броузеры могут делать удивительные вещи. Они могут служить средствами для просмотра документов, запуска удаленных программ, интерфейсами баз данных, воспроизведения мультимедиа и многого другого. Способность открыть в броузере файл локальной или удаленной страницы *изнутри сценария* предоставляет разнообразные интересные возможности для интерфейса пользователя. Например, система под управлением Python может при необходимости автоматически отображать документацию в кодировке HTML, запуская локальный веб-броузер с соответствующим файлом страницы.[1] Поскольку большинство броузеров умеет отображать графику, запускать звуковые файлы и видеоклипы, открытие таких файлов в броузере позволяет сценариям легко справляться с мультимедиа.

Последний сценарий, приводимый в данной главе, менее претенциозен, чем *Launcher.py*, но в такой же мере может быть многократно использован: *LaunchBrowser.py* пытается предоставить *переносимый* интерфейс для запуска веб-броузера. Поскольку техника запуска броузера зависит от платформы, этот сценарий предоставляет интерфейс, в котором эти различия должны быть скрыты от вызывающего. После запуска броузер выполняется как самостоятельная программа, и в нем можно открыть локальный файл или удаленную web-страницу.

Покажем, как это действует. Поскольку большинство броузеров может быть запущено из командной строки оболочки, этот сценарий просто строит надлежащую командную строку и выполняет ее. Например, для запуска броузера Netscape в Linux выполняется команда оболочки вида `netscape url`, где *url* начинается с «file://» для локальных файлов и с «http://» для доступа к удаленным страницам (это соглашения для URL,

Поиск программ в Windows

Согласно совету гуру в Python под Windows, определить местоположение интерпретатора Python в Windows можно с помощью такого специфичного для платформы кода:

```
import _winreg
try:
   keyname = "SOFTWARE\\Microsoft\\Windows\\" +
             "CurrentVersion\\AppPaths\\python.exe"
   pyexe   =_winreg.QueryValue(
            _winreg.HKEY_LOCAL_MACHINE, keyname)
except _winreg.error:
    # not found
```

Этот код использует модуль `_winreg` (появившийся в Python 1.6), для нахождения интерпретатора Python используется модуль `_winreg` (появившийся в Python 1.6), если Python был корректно установлен. Такого рода код будет работать и для других правильно установленных приложений (например, броузеров), но не для файлов некоторых других типов (например, сценариев Python). Кроме того, он слишком специфичен для Windows, чтобы рассказывать о нем подробнее в данной книге; подробности смотрите в ресурсах для Windows.

[1] Например, в панели GUI демонстационной программы PyDemos, с которой мы встретимся в главе 8 «Обзор Tkinter, часть II», есть кнопки, нажатие которых автоматически открывает в броузере веб-страницы, относящиеся к данной книге – сайт издательства, домашнюю страницу Python, мои файлы с исправлениями и т. д.

с которыми мы более подробно познакомимся ниже в главе 12). В Windows того же результата можно добиться командой оболочки вида `start url`. Вот некоторые важные замечания по специфике платформ:

Windows

В Windows сценарий открывает броузер с помощью команды DOS *start* или ищет и запускает броузер с помощью вызова `os.spawnv`. На этой платформе броузер обычно можно открыть с помощью простых команд *start* (например, `os.system("start xxx.html")`). К сожалению, *start* полагается на установленные на машине ассоциации для имен файлов веб-страниц, выбирает броузер в соответствии с этими ассоциациями и имеет ограниченную длину командной строки, которую сценарий может превысить в случае длинных путей к локальным файлам или адресов удаленных страниц.

Поэтому сценарий прибегает к запуску явно указанного броузера с помощью `os.spawnv`, если он запрошен или необходим. Для этого он должен найти полный путь к исполняемому файлу броузера. Поскольку нельзя рассчитывать, что пользователь добавил его в системный путь поиска PATH (или исходный код сценария), сценарий ищет подходящий броузер средствами модуля `Launcher` как в каталогах PATH, так и в обычных местах, куда устанавливаются исполняемые файлы в Windows.

Unix-подобные платформы

На других платформах сценарий использует `os.system` и установку системной переменной PATH на соответствующей машине. Он просто выполняет командную строку с указанием первого броузера из списка кандидатов, найденных в настройках PATH. Поскольку на платформах типа Unix и Linux значительно более вероятно нахождение броузеров в стандартных каталогах поиска (например, */usr/bin*), сценарий не ищет броузер в других каталогах машины. Обратите внимание на `&` в конце выполняемой командной строки: без этого символа вызовы `os.system` вызывают блокирование сценария на Unix-подобных платформах.

Все это можно легко настроить (в конце концов, это код Python), а для других платформ может потребоваться дополнительная логика. Но на всех моих машинах сценарий делает разумные предположения, которые позволяют мне в целом забыть о специфике платформ, обсуждавшейся выше: я просто всюду вызываю одну и ту же функцию `launchBrowser`. Рассмотрим подробности на примере 4.15.

Пример 4.15. PP2E\LaunchBrowser.py

```
#!/bin/env python
#########################################################################################
# Запуск веб-броузера для просмотра веб-страницы переносимым образом. При выполнении в режиме
# '-live' предполагается наличие связи с Интернетом и открывается страница на удаленном сайте.
# В противном случае, предполагается, что страница задана полным путем к файлу на машине,
# и открывается локальный файл. В Unix/Linux отыскивается первый броузер в $PATH. В Windows
# сценарий сначала пытается выполнить команду DOS "start" или ищет местоположение броузера
# на машине для os.spawnv, проверяя PATH и обычные каталоги с исполняемыми файлами Windows.
# Может понадобиться скорректировать имя/каталоги исполняемого файла броузера, если такой поиск
# окажется неудачным. Проверка производилась только под Win98 и Linux, поэтому для других
# машин может потребоваться добавить код (mac: ic.launcurl(url)?).
#########################################################################################
import os, sys
from Launcher import which, guessLocation      # утилиты поиска файлов
useWinStart = 1                                # 0=игнорировать связи имен файлов
onWindows   = sys.platform[:3] == 'win'
helptext    = "Usage: LaunchBrowser.py [ -file path | -live path site ]"
#броузер    = r'c:\"Program Files"\Netscape\Communicator\Program\netscape.exe'
```

```
# значения по умолчанию
Mode = '-file'
Page = os.getcwd() + '/Internet/Cgi-Web/PyInternetDemos.html'
Site = 'starship.python.net/~lutz'
def launchUnixBrowser(url, verbose=1):           # добавьте свою платформу, если она особая
    tries = ['netscape', 'mosaic', 'lynx']       # задайте очередность предпочтений
    for program in tries:
        if which(program): break                 # найти в $path
    else:
        assert 0, 'Sorry - no browser found'
    if verbose: print 'Running', program
    os.system('%s %s &' % (program, url))        # или fork+exec; предполагает $path
def launchWindowsBrowser(url, verbose=1):
    if useWinStart and len(url) <= 400:        # в Windows: start или spawnv
        try:                                  # spawnv действует, когда команда слишком длинная
            if verbose: print 'Starting'
            os.system('start ' + url)          # сначала попробовать связь имен
            return                             # неудача, если командная строка слишком длинна
        except: pass
    browser = None                               # поиск exe-файла броузера
    tries   = ['IEXPLORE.EXE', 'netscape.exe']  # попробовать explorer, потом netscape
    for program in tries:
        browser = which(program) or guessLocation(program, 1)
        if browser: break
    assert browser != None, 'Sorry - no browser found'
    if verbose: print 'Spawning', browser
    os.spawnv(os.P_DETACH, browser, (browser, url))
def launchBrowser(Mode='-file', Page=Page, Site=None, verbose=1):
    if Mode == '-live':
        url = 'http://%s/%s' % (Site, Page)      # открыть страницу на удаленном сайте
    else:
        url = 'file://%s' % Page                 # открыть страницу на этой машине
    if verbose: print 'Opening', url
    if onWindows:
        launchWindowsBrowser(url, verbose)       # использовать windows start, spawnv
    else:
        launchUnixBrowser(url, verbose)          # предположить $path в unix, linux
if __name__ == '__main__':
    # получить аргументы командной строки
    argc = len(sys.argv)
    if argc > 1:  Mode = sys.argv[1]
    if argc > 2:  Page = sys.argv[2]
    if argc > 3:  Site = sys.argv[3]
    if Mode not in ['-live', '-file']:
        print helptext
        sys.exit(1)
    else:
        launchBrowser(Mode, Page, Site)
```

Запуск броузера из командной строки

Данный модуль можно как выполнять, так и импортировать. При его самостоятельном выполнении на моей машине Windows запускается Internet Explorer. Затребованный файл страницы всегда выводится в новом окне броузера, если используется `os.spawnv`, но в окне броузера, открытого в данный момент (если оно есть) при выполнении команды *start*:

```
C:\...\PP2E>python LaunchBrowser.py
```

```
Opening file://C:\PP2ndEd\examples\PP2E/Internet/Cgi-Web/PyInternetDemos.html
Starting
```

Кажущаяся чудной смесь символов прямой и обратной косой черты в показанном URL нормально воспринимается броузером: выводится окно, показанное на рис. 4.2.

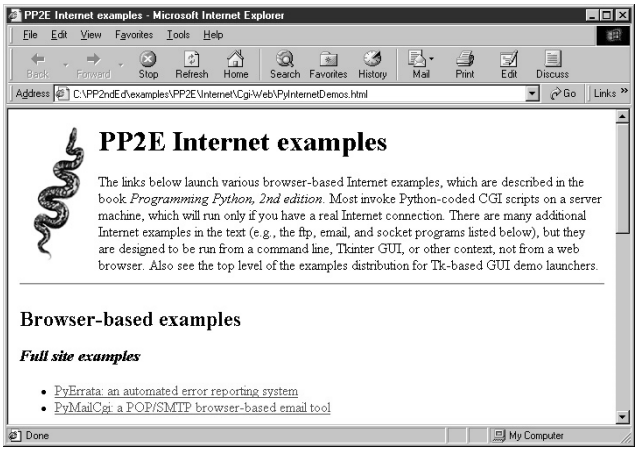


Рис. 4.2. Запуск броузера в Windows с локальным файлом

По умолчанию вызывается команда *start*; чтобы посмотреть, как действует в Windows процедура поиска броузера, установите значение переменной сценария `useWinStart` равным 0. Сценарий будет искать броузер согласно установке PATH, а затем в обычных каталогах Windows, в которые устанавливаются программы, зашитых в коде *Launcher.py*:

```
C:\...\PP2E>python LaunchBrowser.py -file C:\Stuff\Website\public_html\about-pp.html
Opening file://C:\Stuff\Website\public_html\about-pp.html
Looking for IEXPLORE.EXE on ['C:\\WINDOWS', 'C:\\WINDOWS',
'C:\\WINDOWS\\COMMAND', 'C:\\STUFF\\BIN.MKS', 'C:\\PROGRAM FILES\\PYTHON']
Not at C:\WINDOWS\IEXPLORE.EXE
Not at C:\WINDOWS\IEXPLORE.EXE
Not at C:\WINDOWS\COMMAND\IEXPLORE.EXE
Not at C:\STUFF\BIN.MKS\IEXPLORE.EXE
Not at C:\PROGRAM FILES\PYTHON\IEXPLORE.EXE
IEXPLORE.EXE not on system path
Searching for IEXPLORE.EXE in C:\Program Files\Python
Searching for IEXPLORE.EXE in C:\PP2ndEd\examples\PP2E
Searching for IEXPLORE.EXE in C:\Program Files
Spawning C:\Program Files\Internet Explorer\IEXPLORE.EXE
```

Если рассмотреть эти сообщения трассировки, можно заметить, что броузера не было в системном пути поиска и он был обнаружен в локальном подкаталоге *C:\Program Files*, то есть работали функции модуля `Launcher` `which` и `guessLocation`. В соответствии с кодом сценарий сначала ищет Internet Explorer; если вам это не нравится, можно изменить в сценарии список `tries`, чтобы сначала отыскивался Netscape:

```
C:\...\PP2E>python LaunchBrowser.py
Opening file://C:\PP2ndEd\examples\PP2E/Internet/Cgi-Web/PyInternetDemos.html
Looking for netscape.exe on ['C:\\WINDOWS', 'C:\\WINDOWS',
'C:\\WINDOWS\\COMMAND', 'C:\\STUFF\\BIN.MKS', 'C:\\PROGRAM FILES\\PYTHON']
Not at C:\WINDOWS\netscape.exe
```

```
Not at C:\WINDOWS\netscape.exe
Not at C:\WINDOWS\COMMAND\netscape.exe
Not at C:\STUFF\BIN.MKS\netscape.exe
Not at C:\PROGRAM FILES\PYTHON\netscape.exe
netscape.exe not on system path
Searching for netscape.exe in C:\Program Files\Python
Searching for netscape.exe in C:\PP2ndEd\examples\PP2E
Searching for netscape.exe in C:\Program Files
Spawning C:\Program Files\Netscape\Communicator\Program\netscape.exe
```

В данном случае сценарий в конечном счете нашел Netscape в другом каталоге на локальной машине. Помимо автоматического нахождения броузера пользователя, этот сценарий должен быть переносим. При запуске этого файла без каких-либо изменений под Linux запускается локальный броузер Netscape, если он находится в PATH; в противном случае ищутся другие броузеры:

```
[mark@toy ~/PP2ndEd/examples/PP2E]$ python LaunchBrowser.py
Opening file:///home/mark/PP2ndEd/examples/PP2E/Internet/Cgi-
Web/PyInternetDemos.html
Looking for netscape on ['/home/mark/bin', '.', '/usr/bin', '/usr/bin',
'/usr/local/bin', '/usr/X11R6/bin', '/bin', '/usr/X11R6/bin', '/home/mark/
bin', '/usr/X11R6/bin', '/home/mark/bin', '/usr/X11R6/bin']
Not at /home/mark/bin/netscape
Not at ./netscape
Found /usr/bin/netscape
Running netscape
[mark@toy ~/PP2ndEd/examples/PP2E]$
```

У меня установлен Netscape, поэтому запуск сценария на моей машине создает окно, представленное на рис. 4.3 и выведенное в менеджере окон KDE.

Если есть соединение с Интернетом, то можно открывать страницы и на *удаленных* серверах: следующая команда открывает корневую страницу моего сайта на сервере

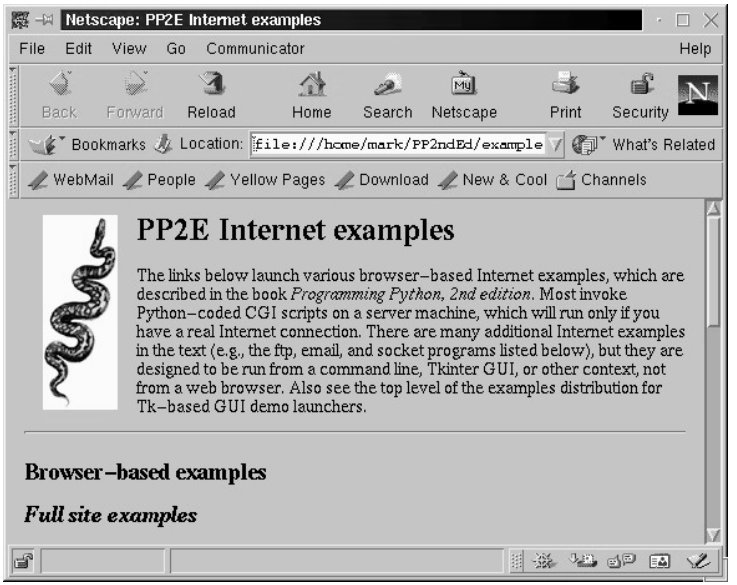


Рис. 4.3. Запуск броузера в Linux

starship.python.net; во время последней моей проверки он располагался где-то на восточном побережье:

```
C:\...\PP2E>python LaunchBrowser.py -live ~lutz starship.python.net
Opening http://starship.python.net/~lutz
Starting
```

В главе 8 мы увидим, что с помощью этого сценария запускаются также примеры для Интернета из системы запуска демонстрационных примеров верхнего уровня: сценарий PyDemos, представленный в той главе, переносимым образом открывает локальные или удаленные файлы веб-страниц с помощью следующей функции обратного вызова для нажатия на кнопку:

```
[File mode]
    pagepath = os.getcwd() + '/Internet/Cgi-Web'
    demoButton('PyErrata',  'Internet-based errata report system',
               'LaunchBrowser.py -file %s/PyErrata/pyerrata.html' % pagepath)
[Live mode]
    site = 'starship.python.net/~lutz'
    demoButton('PyErrata',  'Internet-based errata report system',
               'LaunchBrowser.py -live PyErrata/pyerrata.html ' + site)
```

Запуск броузеров с помощью вызова функций

Другие программы могут, как обычно, порождать такие командные строки *LaunchBrowser.py*, как показанные выше, с помощью таких средств, как `os.system`; но поскольку базовая логика сценария помещена в функцию, ее можно так же просто импортировать и вызывать:

```
>>> from PP2E.LaunchBrowser import launchBrowser
>>> launchBrowser(Page=r'C:\Stuff\Website\Public_html\about-pp.html')
Opening file://C:\Stuff\Website\Public_html\about-pp.html
Starting
>>>
```

При таком вызове `launchBrowser` не сильно отличается от порождения команды *start* в DOS или команды *netscape* в Linux, но функция Python `launchBrowser` предназначена служить переносимым интерфейсом для запуска броузера на разных платформах. Сценарии Python могут использовать этот интерфейс для показа локальных документов HTML в веб-броузерах; на машинах с действующим соединением с Интернетом этот вызов даже позволяет сценариям открывать в броузерах удаленные страницы веб:

```
>>> launchBrowser(Mode='-live', Page='index.html', Site='www.python.org')
Opening http://www.python.org/index.html
Starting
>>> launchBrowser(Mode='-live', Page='~lutz/PyInternetDemos.html',
...                             Site='starship.python.net')
Opening http://starship.python.net/~lutz/PyInternetDemos.html
Starting
```

На моем компьютере первый показанный вызов открывает при необходимости новое окно GUI Internet Explorer, производит дозвон до провайдера через модем и загружает домашнюю страницу Python с *http://www.python.org* как в Windows, так и в Linux – не плохо для одного вызова функции. Второй вызов делает то же самое, но с демонстрационной веб-страницей, которую мы рассмотрим позднее.

«Мультимедийная феерия» Python

Ранее я говорил о том, что броузеры служат недорогим способом показа мультимедиа. Увы, такого рода вещи лучше всего смотреть в натуре, поэтому я могу лишь показать здесь команды для начального запуска. Например, следующая командная строка и вызов функции выводят на моей машине два изображения GIF в Internet Explorer (следите за тем, чтобы использовались полные локальные пути). Результат первого из них приведен на рис. 4.4.

Рис. 4.4. Запуск броузера с графическим файлом

```
C:\...\PP2E>python LaunchBrowser.py -file C:\PP2ndEd\examples\PP2E\Gui\gifs\hills.gif
Opening file://C:\PP2ndEd\examples\PP2E\Gui\gifs\hills.gif
Starting
C:\temp>python
>>> from LaunchBrowser import launchBrowser
>>> launchBrowser(Page=r'C:\PP2ndEd\examples\PP2E\Gui\gifs\mp_lumberjack.gif')
Opening file://C:\PP2ndEd\examples\PP2E\Gui\gifs\mp_lumberjack.gif
Starting
```

Следующая командная строка и вызов открывают на моей машине звуковой файл *sousa.au*; вызов сначала загружает файл с *http://www.python.org*. Если все пройдет хорошо, они и на вашем компьютере сыграют тему из Monty Python:

```
C:\PP2ndEd\examples>python LaunchBrowser.py -file C:\PP2ndEd\examples\PP2E\Internet\Ftp\sousa.au
Opening file://C:\PP2ndEd\examples\PP2E\Internet\Ftp\sousa.au
Starting
>>> launchBrowser(Mode='-live',
...                Site='www.python.org',
...                Page='ftp/python/misc/sousa.au',
...                verbose=0)
>>>
```

Конечно, в Windows можно просто передать эти имена файлов запущенной команде *start* или запустить подходящую обрабатывающую программу непосредственно, скажем, с помощью `os.system`. Но открытие этих файлов в броузере является более переносимым подходом: не нужно хранить набор обрабатывающих файл программ для каждой платформы. При условии, что ваши сценарии используют переносимое средство запуска броузера типа `LaunchBrowser`, не нужно заботиться даже о том, какие броузеры используются на каждой платформе.

В завершение я хочу отметить, что LaunchBrowser знает те броузеры, которые склонен использовать я. Например, в Windows он пытается найти Internet Explorer, а потом уже Netscape, и предпочитает Netscape, а не Mosaic или Lynx в Linux, но ничто не мешает вам изменить этот выбор в своем экземпляре сценария. В действительности LaunchBrowser и Launcher делают некоторые эвристические предположения при поиске файлов, которые могут годиться не для каждого компьютера. Не стесняйтесь «ломать» приведенные примеры: в конце концов, это же Python.

Все пресмыкающиеся мыслят одинаково

Постскриптум. Приблизительно через год после того, как я написал сценарий LaunchBrowser, в версии Python 2.0 объявился новый стандартный библиотечный модуль, служащий аналогичной цели: `webbrowser.open(url)` тоже пытается обеспечить переносимый интерфейс для запуска броузеров из сценариев. Этот модуль сложнее, но, похоже, поддерживает больше вариантов, чем представленный здесь сценарий LaunchBrowser (например, броузеры Macintosh непосредственно поддерживаются с помощью вызова Mac `ic.launcurl(url)` – я бы тоже добавил этот вызов в LaunchBrowser, будь у меня под рукой Mac). Подробности смотрите в руководстве по библиотеке версии 2.0 и выше.

Перед самой сдачей книги в печать я наткнулся еще на один сценарий под именем *FixTk.py* в подкаталоге *lib-tk* библиотеки исходного кода Python; по крайней мере в Python 1.5.2 этот сценарий пытается найти в Windows DLL Tcl/Tk 8.0 путем проверки обычных каталогов для установки программ, чтобы позволить программам Python/Tkinter работать без установки путей Tcl/Tk в PATH. Он не просматривает рекурсивно деревья каталогов, как представленный в этой главе модуль Launcher, и может исчезнуть к тому моменту, когда вы это читаете (Tk копируется в собственный каталог установки Python начиная с Python 2.0), но по духу он близок к некоторым инструментам из последнего раздела данной главы.

5

Более крупные системные примеры, часть 2

«Грепы гнева»[1]

В этой главе будет продолжено изучение конкретных случаев системного программирования. Теперь мы займемся сценариями Python, выполняющими более сложные виды обработки файлов и каталогов. В примерах данной главы решаются такие задачи системного уровня, как преобразование файлов, сравнение и копирование каталогов и поиск строк в файлах и каталогах – задача, известная как «grepping».

С большинством применяемых в этих сценариях инструментов мы ознакомились в главе 2 «Системные инструменты». На этот раз стоит задача показать эти инструменты в действии, в контексте более полезных и реалистичных программ. Как и в предыдущей главе, скрытой подзадачей большинства приведенных здесь примеров является изучение таких технологий программирования на Python, как ООП и инкапсуляция.

Исправление концов строк в формате DOS

Когда я писал первое издание этой книги, для каждого примера на компакт-диске находилось два файла – один с маркерами конца строки в формате Unix и другой с маркерами конца строки в формате DOS. Предполагалось, что это могло облегчить просмотр и редактирование файлов на каждой из платформ. Читатель должен был просто скопировать дерево каталогов, предназначенных для его платформы, на свой жесткий диск и не обращать внимания на второе дерево.

Если вы читали главу 2, то знаете, в чем здесь проблема: DOS (а следовательно, и Windows) помечает конец строки в текстовом файле двумя символами: `\r\n` (возврат каретки, перевод строки), но Unix использует только один `\n`. Большинству современных текстовых редакторов это безразлично – они успешно отображают текстовые файлы, закодированные в любом из этих форматов. Однако некоторые инструменты менее снисходительны. Я по-прежнему иногда вижу странные символы `\r` при просмотре файлов DOS под Unix или целый файл в одной строке при просмотре файлов Unix в DOS (например, это делает Блокнот Windows).

Поскольку такая неприятность встречается редко, а забыть о необходимости согласовать два разных дерева примеров легко, я принял для второго издания иную политику: поставляется один экземпляр примеров (в формате DOS) вместе с переносимым средством конвертирования из одного формата конца строки в другой.

Главным препятствием, конечно, служит необходимость предоставления переносимого и простого в использовании конвертера – такого, который готов к использова-

---

[1] «Греп» – жаргонное слово, произошедшее от английского «grep». Последнее, в свою очередь, также является жаргонным словом, означающим поиск чего-либо с помощью регулярных выражений. – *Примеч. ред.*

нию почти на любом компьютере без редактирования или перекомпиляции. На некоторых платформах Unix есть команды, вроде *fromdos* и *dos2unix*, но они не всюду есть даже на Unix. Пакетные файлы DOS и сценарии csh справились бы с заданием в Windows и Unix соответственно, но ни одно из этих решений не работает одновременно на двух платформах.

К счастью, это не относится к Python. Сценарии, представленные в примерах 5.1, 5.3 и 5.4, преобразуют маркеры конца строки между форматами DOS и Unix. Они преобразуют отдельный файл, каталог файлов и дерево каталогов файлов. В данном разделе мы кратко рассмотрим каждый из этих трех сценариев и выделим некоторые системные средства, которые в них применяются. Каждый из них использует код предшествующего сценария, по сравнению с которым обладает более высокими возможностями обработки.

Пример 5.4, последний из трех, служит тем самым переносимым средством конвертирования, которое мне было нужно: он преобразует маркеры концов строк в целом дереве примеров за один шаг. Так как он написан на чистом Python, то без изменений работает и в DOS, и в Unix. Если у вас установлен Python, этот конвертер строк может стать единственным, который вам когда-либо понадобится.

Преобразование концов строк в одном файле

Эти три сценария намеренно разрабатывались поэтапно, чтобы сначала можно было сосредоточиться на правильном конвертировании перевода строки и только потом заняться логикой каталогов и обхода деревьев. Исходя из такого плана пример 5.1 занимается только задачей преобразования строк в одном текстовом файле.

Пример 5.1. P2E\PyTools\fixeoln_one.py

```
############################################################################################
# Использование: "python fixeoln_one.py [tounix|todos] filename". Преобразует концы строк
# в одном текстовом файле, имя которого указано в командной строке, в целевой формат
# (tounix или todos). Конвертеры _one,  _dir и _all используют функцию convert этого примера.
# convertEndlines изменяет концы строк только при необходимости: строки, уже имеющие целевой
# формат, сохраняются неизменными, поэтому можно конвертировать файл больше одного раза любым
# из 3 сценариев fixeoln. Замечание: для работы сценария под Windows необходимо использовать
# двоичный режим открытия файла, иначе в текстовом режиме по умолчанию автоматически удаляются
# \r при чтении и добавляются лишние \r для каждого \n при записи; формат Mac
# не поддерживается; PyTools\dumpfile.py показывает необработанные байты;
############################################################################################
import os
listonly = 0                                    # 1=показать изменяемый файл, не переписывая его
def convertEndlines(format, fname):               # преобразовать один файл
    if not os.path.isfile(fname):                 # todos:  \n   => \r\n
        print 'Not a text file', fname            # tounix: \r\n => \n
        return                                    # пропустить имена каталогов
    newlines = []
    changed  = 0
    for line in open(fname, 'rb').readlines():  # использовать двоичные режимы для ввода/вывода
        if format == 'todos':                         # иначе \r теряется в Win
            if line[-1:] == '\n' and line[-2:-1] != '\r':
                line = line[:-1] + '\r\n'
                changed = 1
        elif format == 'tounix':                      # избежать IndexError
            if line[-2:] == '\r\n':                   # срезы масштабируются
                line = line[:-2] + '\n'
                changed = 1
```

```
            newlines.append(line)
        if changed:
            try:                                    # может оказаться только для чтения
                print 'Changing', fname
                if not listonly: open(fname, 'wb').writelines(newlines)
            except IOError, why:
                print 'Error writing to file %s: skipped (%s)' % (fname, why)
    if __name__ == '__main__':
        import sys
        errmsg = 'Required arguments missing: ["todos"|"tounix"] filename'
        assert (len(sys.argv) == 3 and sys.argv[1] in ['todos', 'tounix']), errmsg
        convertEndlines(sys.argv[1], sys.argv[2])
        print 'Converted', sys.argv[2]
```

Этот сценарий достаточно прост, что довольно обычно для системных утилит; он полагается, в основном, на методы встроенного файлового объекта. Исходя из заданных флага формата и имени файла он загружает файл в список строк с помощью метода `readlines`, при необходимости преобразует входные строки в целевой формат и записывает результат обратно в файл с помощью метода `writelines`, если какие-либо строки были изменены:

```
C:\temp\examples>python %X%\PyTools\fixeoln_one.py tounix PyDemos.pyw
Changing PyDemos.pyw
Converted PyDemos.pyw

C:\temp\examples>python %X%\PyTools\fixeoln_one.py todos PyDemos.pyw
Changing PyDemos.pyw
Converted PyDemos.pyw

C:\temp\examples>fc PyDemos.pyw %X%\PyDemos.pyw
Comparing files PyDemos.pyw and C:\PP2ndEd\examples\PP2E\PyDemos.pyw
FC: no differences encountered

C:\temp\examples>python %X%\PyTools\fixeoln_one.py todos PyDemos.pyw
Converted PyDemos.pyw

C:\temp\examples>python %X%\PyTools\fixeoln_one.py toother nonesuch.txt
Traceback (innermost last):
  File "C:\PP2ndEd\examples\PP2E\PyTools\fixeoln_one.py", line 45, in ?
    assert (len(sys.argv) == 3 and sys.argv[1] in ['todos', 'tounix']), errmsg
AssertionError: Required arguments missing: ["todos"|"tounix"] filename
```

Здесь первая команда преобразует файл в формат конца строки Unix (*tounix*), а вторая и четвертая преобразуют его в соответствии с соглашениями DOS – независимо от платформы, на которой выполняется сценарий. Чтобы облегчить типичное использование, преобразованный текст записывается обратно в файл *в том же месте*, а не в заново создаваемый выходной файл. Обратите внимание, что в имени файла сценария содержится «_», а не «–»; так как файл предназначен для выполнения в качестве сценария и импортирования в качестве библиотеки, его имя должно переводиться в допустимое имя *переменной* Python в импортирующей программе (*fixeoln-one.py* не может выступать в обеих ролях).

Во всех примерах данной главы, которые изменяют файлы в деревьях каталогов, используемые при тестировании каталоги *C:\temp\examples* и *C:\temp\cpexamples* являются полными копиями реального корневого каталога примеров *PP2E*. Я не всегда привожу команды копирования, используемые для создания этих проверочных каталогов (по крайней мере, пока мы не напишем собственные команды на Python).

Кидаем байты и проверяем результаты

Команда DOS для сравнения файлов *fc* в предшествующем диалоге подтверждает преобразование, но для лучшей проверки результатов выполнения этого сценария Python я написал другой сценарий, представленный в примере 5.2.

Пример 5.2. PP2E\PyTools\dumpfile.py

```
import sys
bytes = open(sys.argv[1], 'rb').read()
print '-'*40
print repr(bytes)
print '-'*40
while bytes:
    bytes, chunk = bytes[4:], bytes[:4]        # показать по 4 байта для строки
    for c in chunk: print oct(ord(c)), '\t',   # показать двоичное значение в восьмеричном виде
    print
print '-'*40
for line in open(sys.argv[1], 'rb').readlines():
    print repr(line)
```

Чтобы ясно представить содержимое файла, этот сценарий открывает его в двоичном режиме (с целью подавления автоматического преобразования перевода строки), выводит его необработанное содержимое (`bytes`) все целиком, выводит в восьмеричном виде ASCII-коды содержимого по четыре байта на строку и показывает необработанные строки. Используем его для трассировки конвертирования. Прежде всего, возьмем простой текстовый файл, чтобы не мучиться, продираясь через байты:

```
C:\temp>type test.txt
a
b
c

C:\temp>python %X%\PyTools\dumpfile.py test.txt
----------------------------------------
'a\015\012b\015\012c\015\012'
----------------------------------------
0141    015     012     0142
015     012     0143    015
012
----------------------------------------
'a\015\012'
'b\015\012'
'c\015\012'
```

Здесь файл *test.txt* имеет формат конца строки DOS – escape-последовательность `\015\012`, выводимая сценарием `dumpfile`, является просто маркером конца строки DOS `\r\n` в формате восьмеричных escape-кодов символов. Теперь преобразование в формат Unix заменит все маркеры DOS `\r\n` на один `\n` (`\012`), как обещано:

```
C:\temp>python %X%\PyTools\fixeoln_one.py tounix test.txt
Changing test.txt
Converted test.txt

C:\temp>python %X%\PyTools\dumpfile.py test.txt
----------------------------------------
'a\012b\012c\012'
----------------------------------------
0141    012     0142    012
0143    012
```

```
----------------------------------------
'a\012'
'b\012'
'c\012'
```

А преобразование обратно в DOS восстановит формат исходного файла:

```
C:\temp>python %X%\PyTools\fixeoln_one.py todos test.txt
Changing test.txt
Converted test.txt

C:\temp>python %X%\PyTools\dumpfile.py test.txt
----------------------------------------
'a\015\012b\015\012c\015\012'
----------------------------------------
0141    015     012     0142
015     012     0143    015
012
----------------------------------------
'a\015\012'
'b\015\012'
'c\015\012'
C:\temp>python %X%\PyTools\fixeoln_one.py todos test.txt    # не делает изменений
Converted test.txt
```

Неразрушающие преобразования

Обратите внимание, что последняя выполненная команда не вывела сообщения «Changing», потому что в действительности в файле не было сделано никаких изменений (он уже имел формат DOS). Так как команда достаточно сообразительна, чтобы не конвертировать строку, уже имеющую заданный формат, ее можно безбоязненно выполнить с файлом, даже если не знать, в каком формате он сейчас находится. Более наивная логика преобразования оказалась бы проще, но использовать ее повторно было бы нельзя. Например, вызов `string.replace` можно использовать для преобразования `\n` из Unix в `\r\n` (`\015\012`) из DOS, но лишь один раз:

```
>>> import string
>>> lines = 'aaa\nbbb\nccc\n'
>>> lines = string.replace(lines, '\n', '\r\n')         # OK:добавлен \r
>>> lines
'aaa\015\012bbb\015\012ccc\015\012'
>>> lines = string.replace(lines, '\n', '\r\n')         # плохо: удвоенный \r
>>> lines
'aaa\015\015\012bbb\015\015\012ccc\015\015\012'
```

Такая логика может легко испортить файл, если применить ее к нему дважды.[1] Однако, чтобы действительно разобраться в том, как сценарий решает эту проблему, необходимо пристальнее взглянуть на использование в нем срезов и двоичного режима открытия файлов.

---

[1] На самом деле посмотрите файлы *old_todos.py*, *old_tounix.py* и *old_toboth.py* в каталоге *PyTools* CD с примерами, и вы найдете полную прежнюю реализацию, основанную на `string.replace`. Ее можно было использовать повторно для преобразований to-Unix, но не для преобразований to-DOS (только последние могут добавлять символы). Для замены были разработаны представленные здесь сценарии `fixeoln` – после того как я обжегся, выполнив дважды преобразования to-DOS.

Получение срезов строк, выходящих за границы

Этот сценарий использует тонкие особенности поведения строк при разрезании, чтобы получать части строк, не зная их размеров. Например:

- Выражение `line[-2:]` возвращает два последних символа в конце строки (либо один или ноль символов, если в строке нет хотя бы двух символов).
- Срез вида `line[-2:-1]` возвращает предпоследний символ (или пустую строку, если строка короткая и в ней нет предпоследнего символа).
- Операция `line[:-2]` возвращает все символы строки, за исключением двух последних (или пустую строку, если в строке меньше трех символов).

Поскольку срезы, выходящие за границы, изменяют величины границ среза до допустимых, не требуется помещать в сценарий дополнительные проверки длины строки, гарантирующие присутствие в ней символов конца строки. Например:

```
>>> 'aaaXY'[-2:], 'XY'[-2:], 'Y'[-2:], ''[-2:]
('XY', 'XY', 'Y', '')

>>> 'aaaXY'[-2:-1], 'XY'[-2:-1], 'Y'[-2:-1], ''[-2:-1]
('X', 'X', '', '')
```

Преобразование строк в Macintosh

Функция `convertEndlines` в таком виде вообще не поддерживает одиночные `\r`, являющиеся терминаторами строк в Macintosh. Она не конвертирует *в* терминаторы Macintosh из формата DOS или Unix (`\r\n` и `\n` в `\r`) и не конвертирует терминаторы *из* формата Macintosh в формат DOS или Unix (`\r` в `\r\n` или `\n`). Файлы в формате Mac проходят в этом сценарии преобразования «todos» и «tounix» без изменений (найдите причины в коде сами). Я не использую Mac, но, возможно, это делают некоторые читатели.

Так как добавление поддержки Mac усложнило бы этот код и так как я не люблю публиковать в книгах код, который не был мною тщательно протестирован, я оставляю такое расширение в качестве упражнения пользователям Python на Mac, если они есть среди читателей. Подсказку для возможной реализации ищите в файле *PP2E\PyTools\fixeoln_one_mac.py* на прилагаемом CD. При выполнении под Windows этот сценарий осуществляет преобразование в Mac:

```
C:\temp>python %X%\PyTools\fixeoln_one_mac.py tomac test.txt
Changing test.txt
Converted test.txt
C:\temp>python %X%\PyTools\dumpfile.py test.txt
----------------------------------------
'a\015b\015c\015'
----------------------------------------
0141    015     0142    015
0143    015
----------------------------------------
'a\015b\015c\015'
```

но не может преобразовать файлы из формата Mac в формат Unix или DOS, потому что метод `readlines` не обрабатывает простой `\r` как перевод строки на этой платформе. Что касается Windows, то последняя строка вывода является единственной строкой файла; при преобразовании обратно в DOS в конец ее просто добавляется один символ `\n`.

```
>>> 'aaaXY'[:-2], 'aaaY'[:-1], 'XY'[:-2], 'Y'[:-1]
('aaa', 'aaa', '', '')
```

Если представить себе символы \r и \n вместо X и Y, то становится ясным, как сценарий использует изменение границ срезов.

Еще раз о двоичном режиме открытия файлов

Так как этот сценарий должен быть переносимым в Windows, он открывает файлы в двоичном режиме несмотря на то, что в них хранятся текстовые данные. Как мы видели, при открытии файлов в текстовом режиме в Windows символ \r удаляется из маркеров \r\n при вводе, и символ \r добавляется перед маркерами \n при выводе. Такое автоматическое преобразование позволяет сценариям представлять маркер конца строки как \n на всех платформах. В данном случае это также означает, что сценарий никогда не увидит \r, который ищет, чтобы определить, имеет ли строка кодировку DOS, – символ \r будет выброшен прежде, чем попадет в сценарий:

```
>>> open('temp.txt', 'w').writelines(['aaa\n', 'bbb\n'])
>>> open('temp.txt', 'rb').read()
'aaa\015\012bbb\015\012'
>>> open('temp.txt', 'r').read()
'aaa\012bbb\012'
```

Если бы не двоичный режим, это привело бы к довольно тонкому и некорректному поведению в Windows. Например, если файлы открываются в текстовом режиме, преобразование в режиме «todos» в Windows в действительности удваивало бы символы \r: сценарий преобразовывал бы оставшийся \n в \r\n, что превращалось бы в выводе в \r\r\n!

```
>>> open('temp.txt', 'w').writelines(['aaa\r\n', 'bbb\r\n'])
>>> open('temp.txt', 'rb').read()
'aaa\015\015\012bbb\015\015\012'
```

В двоичном режиме сценарий полностью вводит \r\n, и преобразования не происходит. Двоичный режим нужен и при выводе в Windows, чтобы подавить вставку символов \r; иначе преобразование «tounix» на этой платформе не работало бы.[1]

Если все это слишком сложно, просто запомните, что нужно использовать «b» в строке режима открытия файла, если сценарий должен выполняться под Windows, и тогда как действительно двоичные данные, так и текстовые данные будут обрабатываться так, как они фактически хранятся в файле.

Преобразование символов конца строки во всех файлах каталога

Имея в руках полностью отлаженный конвертер одного файла, легко перейти к тому, чтобы преобразовывать все файлы, находящиеся в одном каталоге. Нужно просто вызвать его с каждым именем файла, возвращенным средством для перечисления содер-

[1] На самом деле все еще хуже. Из-за автоматического удаления и вставки символов \r в текстовом режиме Windows можно было бы просто читать и писать файлы в текстовом режиме для выполнения преобразования строк «todos» при работе под Windows; интерфейс файлов автоматически добавит в вывод отсутствующий \r. Однако при этом не сработают другие режимы использования – преобразование «tounix» в Windows (пропускать \r можно только при двоичной записи) и «todos» при выполнении в Unix (не будет вставки \r). Волшебство не всегда идет на пользу.

жимого каталога. Сценарий в примере 5.3 использует для получения списка подлежащих преобразованию файлов модуль `glob`, знакомый по главе 2.

Пример 5.3. PP2E\PyTools\fixeoln_dir.py

```
##############################################################################
# Использование: "python fixeoln_dir.py [tounix|todos] patterns?". Преобразование конца
# строки во всех текстовых файлах текущего каталога (рекурсии подкаталогов
# не производится). Использует конвертер однофайловой версии _one.
##############################################################################
import sys, glob
from fixeoln_one import convertEndlines
listonly = 0
patts = ['*.py', '*.pyw', '*.txt', '*.cgi', '*.html',    # имена текстовых файлов
         '*.c',  '*.cxx', '*.h',   '*.i',   '*.out',     # в этом пакете
         'README*', 'makefile*', 'output*', '*.note']
if __name__ == '__main__':
    errmsg = 'Required first argument missing: "todos" or "tounix"'
    assert (len(sys.argv) >= 2 and sys.argv[1] in ['todos', 'tounix']), errmsg

    if len(sys.argv) > 2:                # поиск имен: '*' не применяется в dos
        patts = sys.argv[2:]             # хотя в действительности не требуется в linux
    filelists = map(glob.glob, patts)    # поиск имен только в этом каталоге

    count = 0
    for list in filelists:
        for fname in list:
            if listonly:
                print count+1, '=>', fname
            else:
                convertEndlines(sys.argv[1], fname)
            count = count + 1

    print 'Visited %d files' % count
```

В этом модуле определяется список `patts`, содержащий шаблоны имен файлов, соответствующих всем видам текстовых файлов, встречающихся в дереве каталогов примеров книги; каждый шаблон передается встроенному вызову `glob.glob` через `map`, чтобы быть отдельно превращенным в список соответствующих файлов. Вот почему в конце находятся вложенные циклы `for` – внешний цикл обходит все результирующие списки `glob`, а внутренний цикл обходит все имена внутри каждого списка. Если это непонятно, попробуйте сделать вызов `map` интерактивно:

```
>>> import glob
>>> map(glob.glob, ['*.py', '*.html'])
[['helloshell.py'], ['about-pp.html', 'about-pp2e.html', 'about-ppr2e.html']]
```

Этот сценарий требует задания флага конвертирования в командной строке и предполагает, что выполняется в том же каталоге, где находятся файлы, подлежащие конвертированию; сделайте *cd* в тот каталог, который нужно конвертировать, прежде чем запускать сценарий (или измените его так, чтобы он принимал в качестве аргумента и имя каталога):

```
C:\temp\examples>python %X%\PyTools\fixeoln_dir.py tounix
Changing Launcher.py
Changing Launch_PyGadgets.py
Changing LaunchBrowser.py
...здесь удалены строки...
Changing PyDemos.pyw
```

```
Changing PyGadgets_bar.pyw
Changing README-PP2E.txt
Visited 21 files

C:\temp\examples>python %X%\PyTools\fixeoln_dir.py todos
Changing Launcher.py
Changing Launch_PyGadgets.py
Changing LaunchBrowser.py
...здесь удалены строки...
Changing PyDemos.pyw
Changing PyGadgets_bar.pyw
Changing README-PP2E.txt
Visited 21 files

C:\temp\examples>python %X%\PyTools\fixeoln_dir.py todos    # ничего не изменяет
Visited 21 files

C:\temp\examples>fc PyDemos.pyw %X%\PyDemos.pyw
Comparing files PyDemos.pyw and C:\PP2ndEd\examples\PP2E\PyDemos.pyw
FC: no differences encountered
```

Обратите внимание, что третья команда снова выдала сообщения «Changing». Поскольку для осуществления реального обновления здесь повторно используется функция `convertEndlines` из однофайлового модуля, этот сценарий наследует *возможность повторения* этой функции: можно повторно запускать этот сценарий с одним и тем же каталогом любое число раз. Будут преобразованы только те строки, которые требуют преобразования. Этот сценарий принимает также в командной строке необязательный список имен шаблонов, который переопределяет используемый по умолчанию список изменяемых файлов `patts`:

```
C:\temp\examples>python %X%\PyTools\fixeoln_dir.py tounix *.pyw *.csh
Changing echoEnvironment.pyw
Changing Launch_PyDemos.pyw
Changing Launch_PyGadgets_bar.pyw
Changing PyDemos.pyw
Changing PyGadgets_bar.pyw
Changing cleanall.csh
Changing makeall.csh
Changing package.csh
Changing setup-pp.csh
Changing setup-pp-embed.csh
Changing xferall.linux.csh
Visited 11 files

C:\temp\examples>python %X%\PyTools\fixeoln_dir.py tounix *.pyw *.csh
Visited 11 files
```

Кроме того, заметьте, что функция `convertEndlines` из однофайлового сценария производит первоначальный тест `os.path.isfile` для проверки того, что переданное имя файла представляет собой *файл*, а не каталог: при поиске файлов для конвертирования с помощью шаблонов не исключена возможность того, что наряду с нужными файлами в расширение шаблона могут попасть имена каталогов.

Пользователям Unix и Linux: Unix-подобные оболочки автоматически замещают шаблоны имен файлов типа * в операторах командной строки, прежде чем они попадут в ваш сценарий. Чтобы они попадали в сценарии дословно, их обычно требуется заключать в *кавычки* (например, `"*.py"`).

Сценарий `fixeoln_dir` будет работать, даже если этого не сделать – его вызовы `glob.glob` будут просто находить единственное соответствующее имя файла для уже найденного имени и потому не будут оказывать никакого эффекта:

```
>>> glob.glob('PyDemos.pyw')
['PyDemos.pyw']
```

Однако в оболочке DOS предварительного поиска по шаблону не производится, поэтому использование вызовов `glob.glob` является правильным решением в сценариях, от которых ожидается переносимость.

Преобразование концов строк в целом дереве

И наконец, в примере 5.4 то, чему мы научились, применяется к целому дереву каталогов. Он просто выполняет функцию преобразования файла для каждого имени файла, получаемого логикой обхода дерева. В действительности этот сценарий просто организует вызовы уже отлаженной первоначальной функции `convertEndlines`.

Пример 5.4. PP2E\PyTools\fixeoln_all.py

```
#########################################################################################
# Использование: "python fixeoln_all.py [tounix|todos] patterns?". Найти и преобразовать концы
# строк всех текстовых файлов в каталоге, в котором выполняется этот сценарий, и ниже его
# (том каталоге, где вы находитесь, вводя 'python'). При необходимости пытается использовать
# библиотечный модуль Python find.py либо читает вывод команды find в стиле unix;
# использует установленный по умолчанию список шаблонов имен файлов, если отсутствует аргумент
# с шаблонами. Этот сценарий изменяет только те файлы, которые нуждаются в изменении,
# поэтому его можно без опасения просто запустить из каталога корневого уровня.
#########################################################################################
import os, sys, string
debug    = 0
pyfind   = 0      # вынуждает поиск py
listonly = 0      # 1=только показ результатов поиска
def findFiles(patts, debug=debug, pyfind=pyfind):
    try:
        if sys.platform[:3] == 'win' or pyfind:
            print 'Using Python find'
            try:
                import find                       # использовать код python find.py
            except ImportError:                   # использовать мой, если устарел!
                from PP2E.PyTools import find     # в любом случае можно взять
                                                  # из моего каталога
            matches = map(find.find, patts)       # по умолчанию начальный каталог = '.'
        else:
            print 'Using find executable'
            matches = []
            for patt in patts:
                findcmd = 'find . -name "%s" -print' % patt      # выполнить find
                lines = os.popen(findcmd).readlines()           # удалить концы строк
                matches.append(map(string.strip, lines))        # lambda x: x[:-1]
    except:
        assert 0, 'Sorry - cannot find files'
    if debug: print matches
    return matches
```

```
if __name__ == '__main__':
    from fixeoln_dir import patts
    from fixeoln_one import convertEndlines

    errmsg = 'Required first argument missing: "todos" or "tounix"'
    assert (len(sys.argv) >= 2 and sys.argv[1] in ['todos', 'tounix']), errmsg
    if len(sys.argv) > 2:
        patts = sys.argv[2:]
    matches = findFiles(patts)
    count = 0
    for matchlist in matches:                  # список списков
        for fname in matchlist:                # по одному на шаблон
            if listonly:
                print count+1, '=>', fname
            else:
                convertEndlines(sys.argv[1], fname)
            count = count + 1
    print 'Visited %d files' % count
```

В Windows этот сценарий использует переносимый встроенный инструмент `find.find`, с которым мы познакомились в главе 2 (из Python или самодельный эквивалент),[1] чтобы создать список всех подходящих имен файлов и каталогов в дереве; на других платформах прибегает к порождению хуже переносимой и, возможно, более медленной команды оболочки *find* – чисто в иллюстративных целях.

Когда составлены списки имен путей, этот сценарий просто поочередно преобразует каждый найденный файл с помощью средств из модуля конвертера для одного файла. Вот пример использования сценариев для преобразования дерева примеров из книги в Windows; обратите внимание, что этот сценарий обрабатывает также текущий рабочий каталог (CWD; выполните *cd* в каталог, который нужно преобразовать, прежде чем вводить командную строку) и что Python одинаково воспринимает прямую и обратную косую черту в имени файла программы:

```
C:\temp\examples>python %X%/PyTools/fixeoln_all.py tounix
Using Python find
Changing .\LaunchBrowser.py
Changing .\Launch_PyGadgets.py
Changing .\Launcher.py
Changing .\Other\cgimail.py
...здесь удалено много строк...
Changing .\EmbExt\Exports\ClassAndMod\output.prog1
Changing .\EmbExt\Exports\output.prog1
Changing .\EmbExt\Regist\output
Visited 1051 files

C:\temp\examples>python %X%/PyTools/fixeoln_all.py todos
Using Python find
Changing .\LaunchBrowser.py
Changing .\Launch_PyGadgets.py
Changing .\Launcher.py
```

[1] Вспомните, что исходный каталог выполняемого сценария всегда добавляется в начало `sys.path`, чтобы сценарий мог видеть импортируемые файлы в своем каталоге. Из-за этого сценарий обычно в любом случае загрузит модуль *PP2E\PyTools\find.py* (не из библиотеки Python), просто сказав `import find`; при импорте не обязательно задавать полный путь к пакету. Обработчик `try` и импорт по полному пути полезны здесь только в том случае, если сценарий перемещен в другой каталог. Так как я часто перемещаю файлы, то стремлюсь писать код в расчете на самый худший вариант.

```
Changing .\Other\cgimail.py
...здесь удалено много строк...
Changing .\EmbExt\Exports\ClassAndMod\output.prog1
Changing .\EmbExt\Exports\output.prog1
Changing .\EmbExt\Regist\output
Visited 1051 files

C:\temp\examples>python %X%/PyTools/fixeoln_all.py todos
Using Python find
Not a text file .\Embed\Inventory\Output
Not a text file .\Embed\Inventory\WithDbase\Output
Visited 1051 files
```

Первые две команды преобразуют свыше 1000 файлов, для чего им требуется около 8 секунд реального времени на моей машине 650 МГц с Windows 98; третьей команде требуется только 6 секунд, потому что не нужно обновлять никакие файлы (а на экран выводится меньше сообщений). Однако не относитесь к этим цифрам слишком серьезно: они могут зависеть от загрузки системы, а часть этого времени расходуется, вероятно, на прокрутку вывода на экране.

Взгляд сверху

Этот сценарий и его предшественники поставляются на CD, прилагаемом к книге, в качестве того переносимого средства конвертирования, которое я искал. Чтобы преобразовать все файлы примеров, имеющиеся в дереве, в формат с терминатором строки Unix, скопируйте все дерево примеров *PP2E* в какой-нибудь каталог «examples» на своем жестком диске и введите в оболочке следующие две команды:

```
cd examples/PP2E
python PyTools/fixeoln_all.py tounix
```

Конечно, при этом предполагается, что Python уже установлен (подробности смотрите в файле README на CD), но работать эти команды будут практически на любой используемой сегодня платформе.[1] Чтобы осуществить преобразование обратно в DOS, замените «tounix» на «todos» и выполните команду снова. Я помещаю этот инструмент и на учебный диск для курсов Python, на которых преподаю; для преобразования файлов мы просто вводим:

```
cd Html\Examples
python ..\..\Tools\fixeoln_all.py tounix
```

Привыкнув к командным строкам, вы сможете использовать их в любом контексте. Наконец, чтобы облегчить выполнение преобразования новичкам, в каталог верхнего уровня примеров помещены сценарии *tounix.py* и *todos.py*, по которым можно просто дважды щелкнуть в менеджере файлов; в примере 5.5 показан конвертер «tounix».

Пример 5.5. PP2E\tounix.py

```
#!/usr/local/bin/python
##############################################################################
# Выполнить этот сценарий для преобразования всех текстовых файлов в формат перевода строки
# UNIX/Linux. Делать это нужно тогда, когда при просмотре текстовых файлов этого дистрибутива
# в текстовом редакторе (скажем, в vi) в конце строк видны странные символы '\r'.
# Этот сценарий преобразует все файлы в каталоге примеров и вложенных в него каталогах
```

[1] Кроме, вероятно, Mac – смотрите врезку «Преобразование строк в Macintosh» выше в этой главе. Для конвертирования в формат Mac попробуйте заменить в сценарии импорт `fixeoln_one`, чтобы загружался `fixeoln_one_mac`.

```
# и конвертирует только те файлы, которые еще не преобразованы (его можно выполнять
# несколько раз). Поскольку это сценарий Python, который запускает  другой сценарий Python,
# для выполнения этой программы необходимо сначала установить Python;
# затем из командной строки системы (например, окна xterm) выполнить cd в каталог,
# в котором находится этот сценарий, и ввести "python tounix.py".  Можно также просто
# щелкнуть по значку этого файла в менеджере файлов, если ему известно, что такое файлы '.py'.
#############################################################################################

import os
prompt = """
This program converts all text files in the book
examples distribution to UNIX line-feed format.
Are you sure you want to do this (y=yes)? """

answer = raw_input(prompt)
if answer not in ['y', 'Y', 'yes']:
    print 'Cancelled'
else:
    os.system('python PyTools/fixeoln_all.py tounix')
```

Этот сценарий решает проблему удобства с точки зрения *конечного пользователя*, но на удобства для *программиста*, столь же важные для систем, которые будут читаться и изменяться другими людьми, влияют другие факторы. Например, код конвертеров для файла, каталога и дерева располагается в отдельных файлах сценариев, но ничто не запрещает объединить их в одну программу, которая в зависимости от структуры аргументов командной строки определяет, какой из трех режимов должен быть запущен. Первый аргумент может быть флагом, который проверяется такой программой:

```
if   mode == '-one':
    ...
elif mode == '-dir':
    ...
elif mode == '-all:
    ...
```

Похоже, однако, что это может запутать сильнее, чем отдельные файлы для каждого режима: обычно гораздо легче испортить сложную командную строку, чем ввести имя файла конкретной программы. Кроме того, образуется путаная смесь глобальных имен и один большой участок кода в конце файла. Обычно чем проще, тем лучше.

Исправление имен файлов DOS

В основе предыдущего сценария лежала функция `findFiles`, умеющая переносимым образом собрать подходящие имена файлов и каталогов из целого дерева по заданному списку шаблонов имен файлов. Она делает не больше, чем встроенный вызов `find.find`, но ее можно улучшить для наших целей. Однако поскольку эта логика была собрана в функции, последняя автоматически становится *многократно используемым* инструментом.

Например, следующий сценарий импортирует и применяет `findFiles` для получения *всех* имен файлов в дереве каталогов, используя шаблон имени `*` (он соответствует любым именам). С помощью этого сценария я решаю старую проблему дерева примеров книги. Имена некоторых файлов, созданных в MS-DOS, были переведены в верхний регистр; например *spam.py* где-то по пути превратился в *SPAM.PY*. Поскольку регистр учитывается и в Python, и на некоторых платформах, операция импорта типа «import spam» иногда не срабатывает, если имена файлов записаны в верхнем регистре.

Чтобы исправить это нарушение во всем дереве примеров, насчитывающем тысячу файлов, я написал и выполнил пример 5.6. Действует он так: для каждого имени файла в дереве проверяется, не записано ли все имя заглавными буквами, и пользователю за консолью задается вопрос, следует ли переименовать файл с помощью вызова `os.rename`**. Для облегчения работы для большинства новых имен предлагается разумное значение по умолчанию – прежнее имя, записанное в нижнем регистре.**

Пример 5.6. PP2E\PyTools\fixnames_all.py

```
##############################################################################
# Использование: "python ..\..\PyTools\fixnames_all.py". Найти в текущем каталоге
# ('.') и вложенных в него все файлы, имена которых полностью записаны в верхнем
# регистре (для каждого из них попросить пользователя ввести новое имя файла);
# используется для нахождения старых имен в верхнем регистре, созданных в MS-DOS
# (на некоторых платформах регистр имеет значение при импорте файлов модулей
# Python). Предостережение: может не работать на машинах, различающих регистр,
# если имена каталогов преобразуются раньше, чем их содержимое - исходное имя каталога
# в путях, возвращаемых find, может перестать существовать; эвристика allUpper
# не действует также для необычных имен файлов, состоящих не из букв (пример: '.');
##############################################################################

import os, string
listonly = 0

def allUpper(name):
    for char in name:
        if char in string.lowercase:    # хотя бы одна буква в нижнем регистре дисквалифицирует
            return 0                # или все в верхнем регистре, цифры или специальные символы
    return 1

def convertOne(fname):
    fpath, oldfname = os.path.split(fname)
    if allUpper(oldfname):
        prompt = 'Convert dir=%s file=%s? (y|Y)' % (fpath, oldfname)
        if raw_input(prompt) in ['Y', 'y']:
            default  = string.lower(oldfname)
            newfname = raw_input('Type new file name (enter=%s): ' % default)
            newfname = newfname or default
            newfpath = os.path.join(fpath, newfname)
            os.rename(fname, newfpath)
            print 'Renamed: ', fname
            print 'to:      ', str(newfpath)
            raw_input('Press enter to continue')
            return 1
    return 0

if __name__ == '__main__':
    patts = "*"                             # смотреть все имена файлов
    from fixeoln_all import findFiles       # использовать функцию поиска
    matches = findFiles(patts)

    ccount = vcount = 0
    for matchlist in matches:               # список списков, по одному для шаблона
        for fname in matchlist:             # имена файлов содержат полные пути,
            print vcount+1, '=>', fname     # включая имена каталогов
            if not listonly:
                ccount = ccount + convertOne(fname)
            vcount = vcount + 1
    print 'Converted %d files, visited %d' % (ccount, vcount)
```

Как и прежде, функция `findFiles` возвращает список простых списков имен файлов, представляющих расширения всех переданных в качестве аргументов шаблонов (в данном случае только один список результатов для шаблона `*`).[1] Для каждого имени файла и каталога в результате в этом сценарии функция `convertOne` предлагает изменить имя; комбинация вызовов `os.path.split` и `os.path.join` переносимым образом вставляет новое имя файла в прежний каталог. Вот как происходит переименование в Windows:

```
C:\temp\examples>python %X%\PyTools\fixnames_all.py
Using Python find
1 => .\.cshrc
2 => .\LaunchBrowser.out.txt
3 => .\LaunchBrowser.py
...
...здесь удалены строки...
...
218 => .\Ai
219 => .\Ai\ExpertSystem
220 => .\Ai\ExpertSystem\TODO
Convert dir=.\Ai\ExpertSystem file=TODO? (y|Y)n
221 => .\Ai\ExpertSystem\__init__.py
222 => .\Ai\ExpertSystem\holmes
223 => .\Ai\ExpertSystem\holmes\README.1ST
Convert dir=.\Ai\ExpertSystem\holmes file=README.1ST? (y|Y)y
Type new file name (enter=readme.1st):
Renamed:  .\Ai\ExpertSystem\holmes\README.1st
to:       .\Ai\ExpertSystem\holmes\readme.1st
Press enter to continue
224 => .\Ai\ExpertSystem\holmes\README.2ND
Convert dir=.\Ai\ExpertSystem\holmes file=README.2ND? (y|Y)y
Type new file name (enter=readme.2nd): readme-more
Renamed:  .\Ai\ExpertSystem\holmes\README.2nd
to:       .\Ai\ExpertSystem\holmes\readme-more
Press enter to continue
...
...здесь удалены строки...
...
1471 => .\todos.py
1472 => .\tounix.py
1473 => .\xferall.linux.csh
Converted 2 files, visited 1473
```

Этот сценарий мог бы просто автоматически преобразовать все файлы, имена которых полностью состоят из букв верхнего регистра, в файлы с именами в нижнем регистре, но это опасно (некоторые имена требуют смешанного регистра). Вместо этого во время обхода дерева запрашивается ввод имени, и показываются результаты переименования.

Перезапись с помощью os.path.walk

Заметьте, однако, что вся мощь поиска по шаблону вызова `find.find` в этом сценарии совершенно не используется. Интерфейс `os.path.walk`, изученный в главе 2, можно ис-

[1] Интересно, что строка `'*'` для списка шаблонов действует здесь так же, как список `['*']`, лишь потому, что строка из одного символа является последовательностью, содержащей саму себя; можете проверить, сравнив в интерактивном режиме результат `map(find.find, '*')` с результатом `map(find.find, ['*'])`.

пользовать с таким же успехом, потому что он всегда обходит *все* файлы в дереве, при этом начальная пауза во время составления списка файлов устраняется (в данном случае эта пауза может быть незаметна, но для больших деревьев она может оказаться значительной). Пример 5.7 представляет эквивалентный вариант этого сценария, который осуществляет обход дерева с помощью модели `walk`, основанной на функциях обратного вызова.

Пример 5.7. PP2E\PyTools\fixnames_all2.py

```
#########################################################################
# Использование: "python ..\..\PyTools\fixnames_all2.py". То же самое, но используется
# интерфейс os.path.walk, а не find.find; Чтобы действовать аналогично простой версии find,
# посещение каталогов откладывается до момента непосредственно перед посещением их содержимого
# (find.find перечисляет имена каталогов перед их содержимым); переименование каталогов тоже
# может оказаться неудачным на платформах, чувствительных к регистру, --walk по-прежнему
# расширяет пути, содержащие прежние имена каталогов;
#########################################################################
import os
listonly = 0
from fixnames_all import convertOne
def visitname(fname):
    global ccount, vcount
    print vcount+1, '=>', fname
    if not listonly:
        ccount = ccount + convertOne(fname)
    vcount = vcount + 1
def visitor(myData, directoryName, filesInDirectory):    # вызывается для каждого каталога
    visitname(directoryName)              # обработать каталог, в котором мы сейчас находимся,
    for fname in filesInDirectory:                       # а здесь - файлы-некаталоги
        fpath = os.path.join(directoryName, fname)       # имена файлов не содержат путей
        if not os.path.isdir(fpath):
            visitname(fpath)
ccount = vcount = 0
os.path.walk('.', visitor, None)
print 'Converted %d files, visited %d' % (ccount, vcount)
```

Эта версия делает то же самое, но обходит на один файл больше (самый верхний корневой каталог) и может посещать каталоги в другом порядке (результаты `os.listdir` не упорядочены). На моем компьютере обе версии выполняются в пределах десятка секунд.[1] Мы еще раз вернемся к этому сценарию, как и к исправлению концов строк сценарием `fixeoln`, в контексте общей иерархии классов для обхода дерева далее в этой главе.

[1] Очень тонкий момент: обе версии данного сценария могут отказать на платформах, учитывающих регистр, если в процессе выполнения станут переименовывать каталоги. Если каталог переименован *раньше*, чем обрабатывается его содержимое (например, *SPAM* переименовывается в *spam*), то последующие ссылки на содержимое этого каталога, использующие прежнее имя (например, *SPAM/filename*), оказываются недействительными на платформах, чувствительных к регистру. Это может произойти в версии `find.find`, поскольку каталоги оказываются в списке результатов *перед* своим содержимым. Это может произойти и в версии `os.path.walk`, потому что прежний путь к каталогу (с исходными именами каталогов) продолжает расширяться на каждом уровне дерева. Я использую этот сценарий только в Windows (DOS), поэтому на практике меня это не коснулось. Способы решения этой проблемы – упорядочение списков результатов поиска, обход деревьев в восходящем порядке, выполнение двух раздельных проходов для файлов и каталогов, создание списка каталогов, которые должны быть переименованы позднее – достаточно сложны, чтобы оставить их для экспериментов читателей. Практическое правило гласит, что изменение названий или структуры в деревьях во время их обхода является рискованным мероприятием.

Поиск в деревьях каталогов

Инженеры любят все менять. В процессе написания этой книги я испытывал почти *непреодолимое* желание перемещать и переименовывать каталоги, переменные и совместно используемые модули в дереве примеров книги, как только мне казалось, что я набрел на более подходящую структуру. На достаточно раннем этапе все было приемлемо, но по мере того как дерево становилось все более запутанным, это превратилось в кошмар сопровождения. Пути к каталогам программ и имена модулей повсюду были записаны в коде – в операциях импорта пакетов, вызовах программ, комментариях, файлах конфигурации и т. д.

Можно, конечно, исправлять все эти ссылки, вручную редактируя все файлы в каталоге и находя в каждом ту информацию, которая изменилась. Однако это настолько утомительно, что для дерева примеров данной книги почти совершенно неосуществимо. Когда я пишу эти слова, дерево примеров содержит 118 каталогов и 1342 файла! (Если хотите сосчитать сами, выполните командную строку `python PyTools/visitor.py 1` в корневом каталоге примеров *PP2E*.) Очевидно, требовалось автоматизировать обновление после производимых изменений.

grep и glob в оболочках и Python

Существует стандартный способ поиска строк в файлах для систем Unix и Linux: программа командной строки *grep* и родственные ей перечисляют все строки в одном или нескольких файлах, содержащих строку или шаблон строки.[1] Учитывая, что оболочки Unix расширяют (то есть, «glob») шаблоны имен файлов автоматически, такая команда, как `grep popen *.py`, будет искать в файлах Python, расположенных в одном каталоге, строку «popen». Вот пример выполнения такой команды в Windows (я установил коммерческую Unix-подобную программу *fgrep* на мой переносной компьютер с Windows 98, потому что мне ее там очень не хватало):

```
C:\...\PP2E\System\Filetools>fgrep popen *.py
diffall.py:# - we could also os.popen a diff (unix) or fc (dos)
dirdiff.py:# - use os.popen('ls...') or glob.glob + os.path.split
dirdiff6.py:    files1 = os.popen('ls %s' % dir1).readlines()
dirdiff6.py:    files2 = os.popen('ls %s' % dir2).readlines()
testdirdiff.py:    expected = expected + os.popen(test % 'dirdiff').read()
testdirdiff.py:        output = output + os.popen(test % script).read()
```

В DOS тоже есть команда для поиска в файлах – *find*, которую не нужно путать с командой *find* в Unix, служащей для обхода каталога:

```
C:\...\PP2E\System\Filetools>find /N "popen" testdirdiff.py

---------- testdirdiff.py
[8]    expected = expected + os.popen(test % 'dirdiff').read()
[15]        output = output + os.popen(test % script).read()
```

То же самое можно делать в сценарии Python, выполняя упомянутую выше команду оболочки с помощью `os.system` или `os.popen` либо сочетая встроенные модули `grep`[2] и

---

[1] На самом деле операция поиска в файлах среди разработчиков, проведших в гетто Unix достаточное время, часто носит разговорное название «grepping».

[2] Так как начиная с версии Python 1.6 модуль grep (как и find) объявлен устаревшим, в современных дистрибутивах среди стандартных модулей его нет. Однако в дистрибутиве Python 2.2 для Windows этот модуль (как и некоторые другие, объявленные устаревшими, в том числе find) пока еще можно найти в каталоге Lib\lib-old.

glob. С модулем glob мы уже встречались в главе 2; он расширяет шаблон имени файла в список строк с соответствующими ему именами файлов (подобно оболочке Unix). В стандартную библиотеку входит также модуль grep, который действует, как команда Unix *grep*: grep.grep выводит строки, содержащие заданный шаблон, из группы файлов. При использовании вместе с glob результат очень похож на действие команды *fgrep*:

```
>>> from grep import grep
>>> from glob import glob
>>> grep('popen', glob('*.py'))
diffall.py:  16: # - we could also os.popen a diff (unix) or fc (dos)
dirdiff.py:  12: # - use os.popen('ls...') or glob.glob + os.path.split
dirdiff6.py:  19:     files1 = os.popen('ls %s' % dir1).readlines()
dirdiff6.py:  20:     files2 = os.popen('ls %s' % dir2).readlines()
testdirdiff.py:   8:     expected = expected + os.popen(test % 'dirdiff')...
testdirdiff.py:  15:         output = output + os.popen(test % script).read()

>>> import glob, grep
>>> grep.grep('system', glob.glob('*.py'))
dirdiff.py:  16: # - on unix systems we could do something similar by
regtest.py:  18:         os.system('%s < %s > %s.out 2>&1' % (program, ...
regtest.py:  23:         os.system('%s < %s > %s.out 2>&1' % (program, ...
regtest.py:  24:         os.system('diff %s.out %s.out.bkp > %s.diffs' ...
```

Модуль grep написан на чистом Python (никакие команды оболочки не вызываются), полностью переносим и принимает в качестве ключа поиска как обычные строки, так и шаблоны, содержащие регулярные выражения (о регулярных выражениях будет сказано далее в этой книге). К сожалению, он также ограничен в двух главных отношениях:

- Он просто выводит на экран совпавшие строки, вместо того чтобы создать список, который можно потом обработать. Можно перехватить его вывод и расщепить его, временно переадресовав sys.stdin в объект (в главе 2 было показано, как это сделать), но это довольно неудобно.[1]
- Более важно, что комбинация grep/glob по-прежнему осуществляет просмотр только одного каталога; как было показано в главе 2, для поиска во всех файлах целого дерева каталогов нужны дополнительные усилия.

В системах Unix второе из этих ограничений можно преодолеть, если запустить команду оболочки *grep* из команды оболочки *find*. Например, командная строка Unix:

```
find . -name "*.py" -print -exec fgrep popen {} \;
```

укажет строки и файлы в текущем каталоге и вложенных в него, в которых упоминается «popen». Если Unix-подобная команда *find* будет присутствовать на всех машинах, которыми вам когда-либо придется пользоваться, то один способ обработки каталогов у вас будет всегда.

---

[1] Из-за присущих ему ограничений модуль grep помечен в Python как устаревший («deprecated») начиная с версии 1.6, и в будущих версиях может совсем исчезнуть. Никогда не предполагалось его превращение в широко применяемое средство многократного использования. Для поиска строк в файлах, каталогах и деревьях используйте другие технологии обхода деревьев, описываемые в данной книге. Из первоначальных Unix-подобных модулей grep, glob и find в библиотеке Python только glob на сегодняшний день не считается устаревшим (смотрите также пользовательскую реализацию find в главе 4 «Более крупные системные примеры, часть 1»).

Удаление файлов с байт-кодом

На некоторых из моих машин я запускал сценарий примера 5.8, чтобы удалить все файлы с байт-кодом *.pyc* из дерева примеров, прежде чем упаковывать или обновлять примеры Python (не исключено, что старые двоичные файлы с байт-кодом не совместимы с новыми версиями Python).

Пример 5.8. PP2E\PyTools\cleanpyc.py

```
##############################################################################
# Найти и удалить все файлы с байт-кодом "*.pyc" в каталоге, в котором выполняется
# этот сценарий, и во вложенных в него; предполагается наличие Unix-подобной
# команды find, поэтому переносимость очень слаба; можно использовать вместо нее
# модуль Python find или просто обойти деревья каталогов в переносимом коде Python;
# параметр find -exec тоже может применить сценарий Python к каждому файлу;
##############################################################################

import os, sys

if sys.platform[:3] == 'win':
    findcmd = r'c:\stuff\bin.mks\find . -name "*.pyc" -print'
else:
    findcmd = 'find . -name "*.pyc" -print'
print findcmd

count = 0
for file in os.popen(findcmd).readlines():          # для всех имен файлов
    count = count + 1                               # добавить \n в конце
    print str(file[:-1])
    os.remove(file[:-1])

print 'Removed %d .pyc files' % count
```

В этом сценарии `os.popen` получает вывод программы `find` из коммерческого пакета, установленного на одном из моих Windows-компьютеров, или от стандартного средства `find`, имеющегося в Linux. Он также *абсолютно непереносим* на Windows-машины, если не установлена коммерческая программа `find`, а ее нет на всех компьютерах у меня дома и на большинстве компьютеров в мире в целом.

Сценарии Python могут повторно использовать инструменты соответствующей оболочки с помощью `os.popen`, но в результате они теряют существенную часть переносимости, предоставляемой Python. Команда Unix *find* доступна далеко не всюду, а кроме того, представляет собой сложный инструмент (фактически слишком сложный, чтобы говорить о нем в этой книге; подробности ищите на страницах руководства по Unix). Как было показано в главе 2, команда оболочки наносит также ущерб производительности, поскольку должна запускать на вашем компьютере новую независимую программу.

Чтобы избавиться от потерь в переносимости и производительности, связанных с вызовом команды *find*, я переписал этот код так, чтобы в нем использовались знакомые нам утилиты `find`, и написал главу 2. Новый сценарий приведен в примере 5.9.

Пример 5.9. PP2E\PyTools\cleanpyc-py.py

```
##############################################################################
# Найти и удалить все файлы с байт-кодом "*.pyc" в каталоге, в котором выполняется этот
# сценарий, и во вложенных в него; при этом используется вызов Python find, в результате
# он переносим на большинство машин; выполните этот сценарий, чтобы удалить .pyc
# из прежних версий Python; перед запуском выполните cd в каталог, который нужно очистить;
##############################################################################
```

```
import os, sys, find              # получить find из PyTools

count = 0
for file in find.find("*.pyc"):   # для всех имен файлов
    count = count + 1
    print file
    os.remove(file)

print 'Removed %d .pyc files' % count
```

Этот сценарий переносим и позволяет избежать издержек, связанных с запуском внешних программ. Но `find` в действительности просто осуществляет поиск в дереве, не позволяя вмешаться в него – если во время обхода дерева каталогов нужно сделать что-то необычное, лучше использовать более ручной подход. Кроме того, вызов `find` должен составить список всех имен и только потом может возвратиться; в очень больших деревьях каталогов это может существенно снизить производительность и потребовать большого расхода памяти. Для моих деревьев это не вызывает проблем, но у вас могут быть другие деревья.

Сценарии Python для поиска в дереве

Чтобы облегчить глобальный поиск на всех платформах, которые могут мне когда-либо встретиться, я написал сценарий Python, который выполняет вместо меня основную работу. В примере 5.10 применяются стандартные средства Python, с которыми мы познакомились в предыдущих главах:

- `os.path.walk` для обхода файлов в каталоге
- `string.find` для поиска строки в тексте файла
- `os.path.splitext` для пропуска файлов с расширениями, характерными для двоичных файлов
- `os.path.join` для переносимого соединения пути каталога и имени файла
- `os.path.isdir` для пропуска путей, указывающих на каталоги, а не файлы

Однако будучи чистым кодом Python, этот файл может одинаково выполняться под Linux и Windows. На самом деле он должен работать на любом компьютере, где установлен Python. Более того, благодаря непосредственному использованию системных вызовов он должен работать быстрее, чем при вызове `op.popen` для порождения команды *find*, которая порождает много команд *grep*.

Пример 5.10. PP2E\PyTools\search_all.py

```
#############################################################
# Использование: "python ..\..\PyTools\search_all.py string". Поиск строки во всех файлах
# текущего каталога и вложенных в него; использует интерфейс os.path.walk
# вместо выполнения find для предварительного составления списка имен;
#############################################################

import os, sys, string
listonly = 0
skipexts = ['.gif', '.exe', '.pyc', '.o', '.a']        # игнорировать двоичные файлы

def visitfile(fname, searchKey):                       # для каждого файла-некаталога
    global fcount, vcount                              # искать строку
    print vcount+1, '=>', fname                        # пропустить защищенные файлы
    try:
        if not listonly:
            if os.path.splitext(fname)[1] in skipexts:
```

```
                print 'Skipping', fname
            elif string.find(open(fname).read(), searchKey) != -1:
                raw_input('%s has %s' % (fname, searchKey))
                fcount = fcount + 1
    except: pass
    vcount = vcount + 1

def visitor(myData, directoryName, filesInDirectory):   # вызывается для каждого каталога
    for fname in filesInDirectory:                      # обработать файлы-некаталоги
        fpath = os.path.join(directoryName, fname)      # имена файлов не содержат пути
        if not os.path.isdir(fpath):                    # myData служит ключом поиска
            visitfile(fpath, myData)

def searcher(startdir, searchkey):
    global fcount, vcount
    fcount = vcount = 0
    os.path.walk(startdir, visitor, searchkey)

if __name__ == '__main__':
    searcher('.', sys.argv[1])
    print 'Found in %d files, visited %d' % (fcount, vcount)
```

В этом файле также используется список аргументов командной строки `sys.argv` и прием с `__name__` для работы в двух режимах. При автономном выполнении в командной строке передается ключ поиска; при импортировании клиенты непосредственно вызывают функцию `searcher`. Например, чтобы найти все вхождения имени каталога «Part2» в дереве примеров (старый каталог, которого теперь нет), выполните в оболочке DOS или Unix такую команду:

```
C:\...\PP2E>python PyTools\search_all.py Part2
1 => .\autoexec.bat
2 => .\cleanall.csh
3 => .\echoEnvironment.pyw
4 => .\Launcher.py
.\Launcher.py has Part2
5 => .\Launcher.pyc
Skipping .\Launcher.pyc
6 => .\Launch_PyGadgets.py
7 => .\Launch_PyDemos.pyw
8 => .\LaunchBrowser.out.txt
.\LaunchBrowser.out.txt has Part2
9 => .\LaunchBrowser.py
.\LaunchBrowser.py has Part2
...
...здесь удалены строки...
...
1339 => .\old_Part2\Basics\unpack2b.py
1340 => .\old_Part2\Basics\unpack3.py
1341 => .\old_Part2\Basics\__init__.py
Found in 74 files, visited 1341
```

Этот сценарий выводит список всех проверяемых им файлов, сообщает о пропущенных файлах (имена с расширениями, перечисленными в переменной `skipexts` и подразумевающими двоичные данные) и останавливается, ожидая нажатия Enter после вывода сообщения о нахождении в файле искомой строки (жирные строчки). Решение, основывавшееся на `find`, не могло так останавливаться; в данном примере это делается тривиально, но `find` не возвращается, пока не будет завершен обход всего дерева. Сценарий `search_all` также работает и при импортировании, но не выводит итого-

вой строки со статистикой (fcount и vcount находятся в модуле и их пришлось бы импортировать):

```
>>> from PP2E.PyTools.search_all import searcher
>>> searcher('.', '-exec')          # найти файлы со строкой '-exec'
1 => .\autoexec.bat
2 => .\cleanall.csh
3 => .\echoEnvironment.pyw
4 => .\Launcher.py
5 => .\Launcher.pyc
Skipping .\Launcher.pyc
6 => .\Launch_PyGadgets.py
7 => .\Launch_PyDemos.pyw
8 => .\LaunchBrowser.out.txt
9 => .\LaunchBrowser.py
10 => .\Launch_PyGadgets_bar.pyw
11 => .\makeall.csh
12 => .\package.csh
.\package.csh has -exec
...последующие строки удалены...
```

Каким бы образом ни запускался этот сценарий, он находит все упоминания некоторой строки в целом дереве каталогов – например, изменившееся в примерах имя файла, объекта или каталога.[1]

Visitor: обобщенный обход деревьев

Имея в своем распоряжении сценарий search_all из примера 5.10, я мог точнее находить файлы, которые было необходимо отредактировать при изменении структуры дерева примеров в книге. Вначале я в одном окне запускал search_all, чтобы отобрать подозрительные файлы, и вручную редактировал каждый из них в другом окне.

Однако довольно скоро и это стало утомительным. Вводить вручную имена файлов в командах редактора – занятие невеселое, особенно если нужно редактировать много файлов. Например, приведенный выше поиск «Part2» возвратил 74 файла. Поскольку есть более интересные занятия, чем 74 раза вручную запускать редактор, я стал думать, как автоматически запускать редактор с каждым подозрительным файлом.

К сожалению, search_all просто выводит полученные результаты на экран. Хотя этот текст можно перехватить и проанализировать, проще может оказаться подход, при котором сеансы редактора прямо порождаются во время поиска, но для этого могут потребоваться большие изменения в коде сценария. Здесь мне пришли в голову две мысли.

Во-первых, стало ясно, что в долгосрочной перспективе легче добавлять функции как *внешние компоненты* в общее средство поиска в каталогах, а не менять код исходного сценария. Редактирование файлов могло быть одним из возможных расширений (не автоматизировать ли и замену текста?) для более общего, настраиваемого и многократно используемого компонента для поиска.

---

[1] Смотрите описание регулярных выражений в главе 18 «Текст и язык». В данном случае сценарий search_all ищет в каждом файле простую строку с помощью string.find, но можно тривиально расширить его так, чтобы выполнялся поиск по шаблону регулярного выражения (грубо говоря, заменить string.find на вызов метода поиска объекта регулярного выражения). Конечно, такое изменение станет гораздо более тривиальным после того, как мы научимся это делать.

Во-вторых, написав несколько утилит для обхода каталогов, я понял, что каждый раз пишу заново код одного и того же типа. Обход можно упростить еще больше, если скрыть детали под оболочкой и облегчить повторное использование. Средство `os.path.walk` полезно, но при его использовании возникают лишние действия (например, присоединение имен каталогов), а основанный на функциях интерфейс не так хорошо поддается настройке, как при использовании классов.

Конечно, обе задачи подсказывают, что для обхода и поиска должна быть использована ОО-структура. Пример 5.11 представляет одну возможную конкретную реализацию этих задач. Он экспортирует общий класс `FileVisitor`, который, в основном, лишь служит оболочкой для `os.path.walk`, облегчающей использование и расширение, а также родовой класс `SearchVisitor`, обобщающий идею поиска в каталоге. Сам по себе `SearchVisitor` делает то же самое, что делал `search_all`, но кроме того, он делает процесс поиска открытым для настройки – какие-то черты его поведения могут модифицироваться путем перегрузки методов в подклассах. Более того, его базовая логика поиска может быть использована везде, где требуется поиск: нужно просто определить подкласс, в котором будут добавлены специфические для поиска расширения.

Пример 5.11. PP2E\PyTools\visitor.py

```
##########################################################################
# Тест: "python ..\..\PyTools\visitor.py testmask [строка]". Использует ООП,
# классы и подклассы для оболочки, скрывающей детали использования os.path.walk
# при обходе и поиске; testmask является битовой маской, биты которой задают
# имеющиеся самопроверки; См. также: подклассы visitor_edit/replace/find/fix*/.py
# и сценарий клиента fixsitename.py в Internet\Cgi-Web;
##########################################################################

import os, sys, string
listonly = 0

class FileVisitor:
    """
    проверяет все файлы, не являющиеся каталогами внутри каталога startDir;
    переопределите visitfile, чтобы установить свой обработчик файлов
    """
    def __init__(self, data=None, listonly=0):
        self.context  = data
        self.fcount   = 0
        self.dcount   = 0
        self.listonly = listonly
    def run(self, startDir=os.curdir):                  # по умолчанию начало ='.'
        os.path.walk(startDir, self.visitor, None)
    def visitor(self, data, dirName, filesInDir):       # вызывается для каждого каталога
        self.visitdir(dirName)                          # сначала этот каталог
        for fname in filesInDir:                        # обработать файлы-некаталоги
            fpath = os.path.join(dirName, fname)        # имена файлов не содержат путей
            if not os.path.isdir(fpath):
                self.visitfile(fpath)
    def visitdir(self, dirpath):                        # вызывается для каждого каталога
        self.dcount = self.dcount + 1                   # переопределить или расширить
        print dirpath, '...'
    def visitfile(self, filepath):                      # вызывается для каждого файла
        self.fcount = self.fcount + 1                   # переопределить или расширить
        print self.fcount, '=>', filepath               # по умолчанию: вывести имя

class SearchVisitor(FileVisitor):
    """
```

```
    искать строку в файлах внутри startDir и его подкаталогов
    """
    skipexts = ['.gif', '.exe', '.pyc', '.o', '.a']     # пропуск двоичных файлов
    def __init__(self, key, listonly=0):
        FileVisitor.__init__(self, key, listonly)
        self.scount = 0
    def visitfile(self, fname):                         # проверка совпадения
        FileVisitor.visitfile(self, fname)
        if not self.listonly:
            if os.path.splitext(fname)[1] in self.skipexts:
                print 'Skipping', fname
            else:
                text = open(fname).read()
                if string.find(text, self.context) != -1:
                    self.visitmatch(fname, text)
                    self.scount = self.scount + 1
    def visitmatch(self, fname, text):                   # обработка совпадения
        raw_input('%s has %s' % (fname, self.context))   # переопределить меня ниже

# логика самотестирования
dolist   = 1
dosearch = 2    # 3=список и поиск
donext   = 4    # при добавлении следующей проверки

def selftest(testmask):
    if testmask & dolist:
       visitor = FileVisitor()
       visitor.run('.')
       print 'Visited %d files and %d dirs' % (visitor.fcount, visitor.dcount)

    if testmask & dosearch:
       visitor = SearchVisitor(sys.argv[2], listonly)
       visitor.run('.')
       print 'Found in %d files, visited %d' % (visitor.scount, visitor.fcount)
if __name__ == '__main__':
    selftest(int(sys.argv[1]))    # e.g., 5 = dolist | dorename
```

Этот модуль служит в основном для экспорта классов, используемых другими программами, но и при самостоятельном выполнении делает кое-что полезное. Если вызвать его как сценарий с одним аргументом «1», он создаст и запустит объект `FileVisitor` и выведет полный список всех файлов и каталогов начиная с того каталога, откуда он вызван, и ниже (то есть «.», текущего рабочего каталога):

```
C:\temp>python %X%\PyTools\visitor.py 1
. ...
1 => .\autoexec.bat
2 => .\cleanall.csh
3 => .\echoEnvironment.pyw
4 => .\Launcher.py
5 => .\Launcher.pyc
6 => .\Launch_PyGadgets.py
7 => .\Launch_PyDemos.pyw
...здесь удалены строки...
479 => .\Gui\Clock\plotterGui.py
480 => .\Gui\Clock\plotterText.py
481 => .\Gui\Clock\plotterText1.py
482 => .\Gui\Clock\__init__.py
.\Gui\gifs ...
483 => .\Gui\gifs\frank.gif
```

```
484 => .\Gui\gifs\frank.note
485 => .\Gui\gifs\gilligan.gif
486 => .\Gui\gifs\gilligan.note
...здесь удалены строки...
1352 => .\PyTools\visitor_fixnames.py
1353 => .\PyTools\visitor_find_quiet2.py
1354 => .\PyTools\visitor_find.pyc
1355 => .\PyTools\visitor_find_quiet1.py
1356 => .\PyTools\fixeoln_one.doc.txt
Visited 1356 files and 119 dirs
```

Если же вызвать этот сценарий с «2» в качестве первого аргумента, он создаст и запустит объект `SearchVisitor`, используя второй аргумент в качестве ключа поиска. Этот формат эквивалентен запуску знакомого сценария *search_all.py*; он останавливается, ожидая нажатия Enter после сообщения о каждом найденном файле (строки, выделенные здесь жирным шрифтом):

```
C:\temp\examples>python %X%\PyTools\visitor.py 2 Part3
. ...
1 => .\autoexec.bat
2 => .\cleanall.csh
.\cleanall.csh has Part3
3 => .\echoEnvironment.pyw
4 => .\Launcher.py
.\Launcher.py has Part3
5 => .\Launcher.pyc
Skipping .\Launcher.pyc
6 => .\Launch_PyGadgets.py
7 => .\Launch_PyDemos.pyw
8 => .\LaunchBrowser.out.txt
9 => .\LaunchBrowser.py
10 => .\Launch_PyGadgets_bar.pyw
11 => .\makeall.csh
.\makeall.csh has Part3
...
...здесь удалены строки...
...
1353 => .\PyTools\visitor_find_quiet2.py
1354 => .\PyTools\visitor_find.pyc
Skipping .\PyTools\visitor_find.pyc
1355 => .\PyTools\visitor_find_quiet1.py
1356 => .\PyTools\fixeoln_one.doc.txt
Found in 49 files, visited 1356
```

Технически при передаче сценарию в качестве первого аргумента «3» он выполняет *оба* объекта, `FileVisitor` и `SearchVisitor` (осуществляется два отдельных обхода). Первый аргумент в действительности используется в качестве битовой маски для выбора одной или более поддерживаемых самопроверок – если бит для теста установлен в двоичном значении аргумента, тест будет выполнен. Поскольку 3 представляется в двоичном виде как 011, выбираются как поиск (010), так и вывод списка (001). В системе, более дружественной пользователю, можно было бы задать символические параметры (например, искать аргументы «–search» и «–list»), но для целей данного сценария достаточно битовых масок.

Война и мир текстовых редакторов

Возможно, вы не знаете, что установка vi в сценарии *visitor_edit.py* означает известный текстовый редактор Unix; существует его версия для Windows, но там этот редактор не является стандартным. Для запуска этого сценария на вашей машине может потребоваться изменить настройку `editor`. Например, в Linux должна работать установка «emacs», а на любой машине Windows должен работать «edit» или «notepad».

В данное время я склонен использовать редактор, который написал на Python (PyEdit), поэтому воевать за редакторы я предоставлю более политически активным читателям. В действительности, если изменить код сценария, чтобы `editor` присваивалось одно из следующих значений:

```
Editor = r'python Gui\TextEditor\textEditor.pyw'
editor = r'start  Gui\TextEditor\textEditor.pyw'
```

то найденный файл будет открыт в GUI переносимого и написанного на чистом Python текстового редактора – закодированного на Python с интерфейсом Tkinter и выполняемого на всех основных платформах GUI. Мы познакомимся с ним в главе 9 «Более крупные примеры GUI». Если вы прочли про команду *start* в главе 3 «Системные средства параллельного выполнения», то знаете, что при первой установке значения editor обход дерева останавливается на время работы редактора, а при второй – нет (будет открыто столько окон PyEdit, сколько найдено файлов).

Однако запуск редактора может оказаться неудачным, если имя пути к его каталогу слишком длинное (вспомните, что в `os.system`, в отличие от `os.spawnv`, есть ограничение на длину). Кроме того, путь к программе *textEditor.pyw* может быть разным в зависимости от места, из которого вы запустили *visitor_edit.py* (то есть от CWD). С последней проблемой можно справиться несколькими способами:

- Задать в качестве префикса пути к сценарию значение переменной окружения `PP2EHOME`, которое можно получить от `os.environ`; для стандартных сценариев установки книги `PP2EHOME` дает абсолютное значение корневого каталога, от которого можно найти путь к редактору.
- Задать в качестве префикса пути `sys.path[0]` и `'../'`, используя то обстоятельство, что первым каталогом для импорта всегда является исходный каталог сценария (см. раздел «Текущий рабочий каталог» в главе 2).
- Создать ярлыки Windows или ссылки Unix для сценария редактора в CWD.
- Осуществить простой поиск сценария с помощью `Launcher.findFirst` или `guessLocation`, описанных в конце главы 4.

Но все это находится за рамками врезки, посвященной политике выбора текстового редактора.

Редактирование файлов в деревьях каталогов

Теперь, после обобщения обхода деревьев и поиска, легко сделать следующий шаг и добавить автоматическое редактирование файла с помощью отдельного совершенно нового компонента. В примере 5.12 определен новый класс `EditVisitor`, который просто изменяет метод `visitmatch` класса `SearchVisitor`, чтобы открывать найденный файл в текстовом редакторе. Да, это законченная программа – что-либо особое нужно делать

только при обработке найденных файлов, и только это поведение должно обеспечиваться; все остальное, касающееся логики обхода и поиска, остается неизменным и наследуется.

Пример 5.12. PP2E\PyTools\visitor_edit.py

```
############################################################################################
# Использование: "python PyTools\visitor_edit.py строка". Добавляет в SearchVisitor
# автоматический запуск редактора как внешнего компонента (подкласса), не меняя кода;
# в этой версии во время обхода автоматически вызывается редактор для каждого файла,
# содержащего заданную строку; можно также задать editor='edit' или 'notepad'
# в Windows; 'vi' и 'edit' выполняются в окне консоли;
# editor=r'python Gui\TextEditor\textEditor.pyw' тоже может работать;
# Замечание: для некоторых редакторов можно сделать еще толковее,
# передавая команду поиска для нахождения первого совпадения;
############################################################################################

import os, sys, string

from visitor import SearchVisitor
listonly = 0

class EditVisitor(SearchVisitor):
    """
    редактировать файлы, содержащие заданную строку в каталоге startDir и его подкаталогах
    """
    editor = 'vi'  # ymmv
    def visitmatch(self, fname, text):
        os.system('%s %s' % (self.editor, fname))

if __name__  == '__main__':
    visitor = EditVisitor(sys.argv[1], listonly)
    visitor.run('.')
    print 'Edited %d files, visited %d' % (visitor.scount, visitor.fcount)
```

При создании и запуске `EditVisitor` текстовый редактор запускается с помощью вызова командной строки `os.system`, который обычно блокирует вызывающего до того момента, когда завершится порожденная программа. На моих машинах при каждом нахождении сценарием соответствующего файла во время обхода запускается текстовый редактор vi в том окне консоли, в котором был запущен сценарий; при выходе из редактора возобновляется обход дерева.

Найдем и отредактируем несколько файлов. При запуске в качестве сценария мы передаем этой программе в качестве аргумента искомую строку (здесь строка «–exec» является ключом поиска, а не флагом опции). Корневой каталог всегда передается методу `run` как «.», текущий каталог выполнения. Сообщения о состоянии обхода, как и прежде, выводятся на консоль, но каждый файл, в котором найдена строка, тут же автоматически открывается в текстовом редакторе. В данном случае редактор запускается восемь раз:

```
C:\...\PP2E>python PyTools\visitor_edit.py -exec
1 => .\autoexec.bat
2 => .\cleanall.csh
3 => .\echoEnvironment.pyw
4 => .\Launcher.py
5 => .\Launcher.pyc
Skipping .\Launcher.pyc
...здесь удалены строки...
1340 => .\old_Part2\Basics\unpack2.py
```

```
1341 => .\old_Part2\Basics\unpack2b.py
1342 => .\old_Part2\Basics\unpack3.py
1343 => .\old_Part2\Basics\__init__.py
Edited 8 files, visited 1343
```

В итоге получается то средство, которое я искал, чтобы упростить сопровождение дерева примеров книги. После значительных изменений в таких вещах, как имена совместно используемых модулей и файлов и каталогов, я запускаю этот сценарий для корневого каталога примеров с соответствующей строкой поиска и нужным образом редактирую все открывающиеся файлы. Мне все же приходится вручную изменять файлы в редакторе, но это зачастую безопаснее, чем вслепую выполнять глобальную замену.

Глобальная замена в деревьях каталогов

Но если уж я затронул этот вопрос, то имея общий класс для обхода дерева, легко написать и подкласс для глобального поиска и замены. В примере 5.13 подкласс `ReplaceVisitor` класса `FileVisitor` модифицирует метод `visitfile` так, чтобы глобально заменять все вхождения одной строки другой строкой во всех текстовых файлах, находящихся в корневом каталоге и ниже. Он также составляет список всех измененных файлов, чтобы можно было просмотреть файлы и проверить автоматически сделанные изменения (можно, например, автоматически вызывать текстовый редактор для каждого измененного файла).

Пример 5.13. PP2E\PyTools\visitor_replace.py

```
################################################################################
# Использование: "python PyTools\visitor_replace.py fromStr toStr". Осуществляет
# глобальные поиск и замену во всех файлах дерева каталогов - заменяет fromStr на toStr
# во всех текстовых файлах; это сильно, но опасно!! visitor_edit.py запускает редактор,
# чтобы можно было проверить и сделать изменения, а потому гораздо безопаснее;
# с помощью CollectVisitor можно просто составить список найденных файлов;
################################################################################
import os, sys, string
from visitor import SearchVisitor
listonly = 0
class ReplaceVisitor(SearchVisitor):
    """
    изменить fromStr на toStr в файлах в каталоге startDir и его подкаталогах;
    список измененных файлов будет в массиве obj.changed после выполнения
    """
    def __init__(self, fromStr, toStr, listonly=0):
        self.changed = []
        self.toStr   = toStr
        SearchVisitor.__init__(self, fromStr, listonly)
    def visitmatch(self, fname, text):
        fromStr, toStr = self.context, self.toStr
        text = string.replace(text, fromStr, toStr)
        open(fname, 'w').write(text)
        self.changed.append(fname)
if __name__  == '__main__':
    if raw_input('Are you sure?') == 'y':
        visitor = ReplaceVisitor(sys.argv[1], sys.argv[2], listonly)
        visitor.run(startDir='.')
        print 'Visited %d files'  % visitor.fcount
        print 'Changed %d files:' % len(visitor.changed)
        for fname in visitor.changed: print fname
```

Для того чтобы выполнить этот сценарий с деревом каталогов, зайдите в нужный каталог и выполните командную строку типа приведенной ниже, задав заменяемую и заменяющую строки. На моей машине обработка дерева с 1354 файлами, из которых 75 нужно изменить, занимает примерно шесть секунд реального времени, если система не слишком занята другими задачами:

```
C:\temp\examples>python %X%/PyTools/visitor_replace.py Part2 SPAM2
Are you sure?y
. ...
1 => .\autoexec.bat
2 => .\cleanall.csh
3 => .\echoEnvironment.pyw
4 => .\Launcher.py
5 => .\Launcher.pyc
Skipping .\Launcher.pyc
6 => .\Launch_PyGadgets.py
...здесь удалены строки...
1351 => .\PyTools\visitor_find_quiet2.py
1352 => .\PyTools\visitor_find.pyc
Skipping .\PyTools\visitor_find.pyc
1353 => .\PyTools\visitor_find_quiet1.py
1354 => .\PyTools\fixeoln_one.doc.txt
Visited 1354 files
Changed 75 files:
.\Launcher.py
.\LaunchBrowser.out.txt
.\LaunchBrowser.py
.\PyDemos.pyw
.\PyGadgets.py
.\README-PP2E.txt
...здесь удалены строки...
.\PyTools\search_all.out.txt
.\PyTools\visitor.out.txt
.\PyTools\visitor_edit.py

[to delete, use an empty toStr]
C:\temp\examples>python %X%/PyTools/visitor_replace.py SPAM ""
```

Это одновременно очень мощный и опасный прием. Если заменяемая строка может обнаружиться в неожиданных местах, запуск определенного здесь объекта `ReplaceVisitor` может разрушить все дерево файлов. С другой стороны, если строка является чем-то очень специфическим, этот объект помогает избежать необходимости автоматически редактировать подозрительные файлы. Например, в главе 12 «Сценарии, выполняемые на сервере» такой подход будет применен для автоматического изменения адресов сайтов в файлах HTML: адреса слишком специфичны, чтобы случайно появиться в других местах.

Составление списка найденных файлов в деревьях каталогов

Приведенные выше сценарии осуществляют поиск и замену в деревьях каталогов с помощью одного и того же базового кода, осуществляющего обход (модуля `visitor`). Предположим, что нужно лишь получить в Python *список* файлов в каталоге, содержащих некоторую строку. Можно было бы выполнить поиск и выделить в выводе сообщения «found». Гораздо проще, однако, создать еще один подкласс `SearchVisitor`, который будет попутно составлять список, как в примере 5.14.

Пример 5.14. PP2E\PyTools\visitor_collect.py

```
##################################################################
# Использование: "python PyTools\visitor_collect.py searchstring". CollectVisitor
# просто составляет список найденных файлов, который можно вывести или использовать для
# дальнейшей обработки (например, замены, автоматического редактирования);
##################################################################
import os, sys, string
from visitor import SearchVisitor
class CollectVisitor(SearchVisitor):
    """
    собрать имена файлов, содержащих заданную строку;
    запустить и затем взять список obj.matches
    """
    def __init__(self, searchstr, listonly=0):
        self.matches = []
        SearchVisitor.__init__(self, searchstr, listonly)
    def visitmatch(self, fname, text):
        self.matches.append(fname)
if __name__  == '__main__':
    visitor = CollectVisitor(sys.argv[1])
    visitor.run(startDir='.')
    print 'Found these files:'
    for fname in visitor.matches: print fname
```

`CollectVisitor` снова осуществляет поиск в дереве, но с новым типом специализации – сбором имен файлов вместо вывода сообщений. Этот класс удобно использовать в других сценариях, которые должны составить список файлов для дальнейшей обработки, но его можно выполнять и самостоятельно, как сценарий:

```
C:\...\PP2E>python PyTools\visitor_collect.py -exec
...
...здесь удалены строки...
...
1342 => .\old_Part2\Basics\unpack2b.py
1343 => .\old_Part2\Basics\unpack3.py
1344 => .\old_Part2\Basics\__init__.py
Found these files:
.\package.csh
.\README-PP2E.txt
.\readme-old-pp1E.txt
.\PyTools\cleanpyc.py
.\PyTools\fixeoln_all.py
.\System\Processes\output.txt
.\Internet\Cgi-Web\fixcgi.py
```

Подавление вывода сообщений о состоянии

Здесь в конце выводится составленный список всех файлов, содержащих строку «–exec». Обратите, однако, внимание, что сообщения о состоянии обхода по-прежнему выводятся по ходу работы (в действительности я удалил здесь около 1600 строк с такими сообщениями!). Для инструмента, который предполагается вызывать из другого сценария, это может оказаться нежелательным побочным эффектом: вывод вызывающего сценария может быть более важным, чем сопутствующие обходу сообщения.

Можно было бы добавить в `SearchVisitor` флаги режима, отключающие вывод сообщений о состоянии, но это слишком сложно. Вместо этого приведем следующие два файла, показывающие, как можно составить список найденных файлов без вывода на

консоль сообщений о состоянии обхода, не изменяя при этом базового исходного кода. Первый, показанный в примере 5.15, просто копирует логику поиска, но без команд вывода. Это создает некоторую избыточность при наличии `SearchVisitor`, но всего лишь в нескольких строках копируемого кода.

Пример 5.15. PP2E\PyTools\visitor_collect_quiet1.py

```
##############################################################
# Похоже на visitor_collect, но без сообщений о состоянии обхода
##############################################################
import os, sys, string
from visitor import FileVisitor, SearchVisitor

class CollectVisitor(FileVisitor):
    """
собрать имена файлов, содержащих заданную строку, молча;
    """
    skipexts = SearchVisitor.skipexts
    def __init__(self, searchStr):
        self.matches = []
        self.context = searchStr
    def visitdir(self, dname): pass
    def visitfile(self, fname):
        if (os.path.splitext(fname)[1] not in self.skipexts and
            string.find(open(fname).read(), self.context) != -1):
            self.matches.append(fname)

if __name__  == '__main__':
    visitor = CollectVisitor(sys.argv[1])
    visitor.run(startDir='.')
    print 'Found these files:'
    for fname in visitor.matches: print fname
```

При выполнении этого класса выводится только содержимое списка найденных файлов в конце работы, во время обхода не выводятся никакие сообщения о состоянии. Благодаря этому такая форма может оказаться более подходящей для инструмента общего назначения, используемого другими сценариями:

```
C:\...\PP2E>python PyTools\visitor_collect_quiet1.py -exec
Found these files:
.\package.csh
.\README-PP2E.txt
.\readme-old-pp1E.txt
.\PyTools\cleanpyc.py
.\PyTools\fixeoln_all.py
.\System\Processes\output.txt
.\Internet\Cgi-Web\fixcgi.py
```

Более интересным и создающим меньше избыточности способом подавления вывода текста во время обхода может оказаться технология переадресации потоков, с которой мы познакомились в главе 2. В примере 5.16 `sys.stdin` устанавливается в объект `NullOut`, который выбрасывает весь выводимый текст во время обхода (его метод `write` не выполняет никаких действий).

Единственное реальное осложнение в такой схеме связано с отсутствием подходящего места для восстановления `sys.stdout` в конце обхода. Мы поместили восстановление в метод деструктора `__del__`, при этом требуется, чтобы клиент удалил обходчика, чтобы восстановить обычный вывод. Если вы предпочитаете менее «магические» интерфейсы, то метод можно вызвать и явным образом.

Пример 5.16. PP2E\PyTools\visitor_collect_quiet2.py

```
#########################################################################
# Похоже на visitor_collect, но без сообщений о состоянии обхода
#########################################################################
import os, sys, string
from visitor import SearchVisitor
class NullOut:
    def write(self, line): pass
class CollectVisitor(SearchVisitor):
    """
    собрать имена файлов, содержащих заданную строку, молча
    """
    def __init__(self, searchstr, listonly=0):
        self.matches = []
        self.saveout, sys.stdout = sys.stdout, NullOut()
        SearchVisitor.__init__(self, searchstr, listonly)
    def __del__(self):
        sys.stdout = self.saveout
    def visitmatch(self, fname, text):
        self.matches.append(fname)
if __name__  == '__main__':
    visitor = CollectVisitor(sys.argv[1])
    visitor.run(startDir='.')
    matches = visitor.matches
    del visitor
    print 'Found these files:'
    for fname in matches: print fname
```

При выполнении этого сценария получается вывод, совпадающий с предыдущим – только имена найденных файлов в конце. А может быть, еще лучше просто запрограммировать и отладить один разговорчивый вспомогательный класс `CollectVisitor` и потребовать, чтобы *клиент* создавал оболочку для вызовов его метода `run` в функции `redirect.redirect`, которую мы написали в примере 2.10?

```
>>> from PP2E.PyTools.visitor_collect import CollectVisitor
>>> from PP2E.System.Streams.redirect import redirect
>>> walker = CollectVisitor('-exec')           # объект для поиска '-exec'
>>> output = redirect(walker.run, ('.',), '')  # функция, аргументы, входные данные
>>> for line in walker.matches: print line     # вывод элементов вписка
...
.\package.csh
.\README-PP2E.txt
.\readme-old-pp1E.txt
.\PyTools\cleanpyc.py
.\PyTools\fixeoln_all.py
.\System\Processes\output.txt
.\Internet\Cgi-Web\fixcgi.py
```

Примененный здесь вызов `redirect` переустанавливает стандартные входной и выходной потоки в объекты типа файл на время выполнения вызова *любой* функции; поэтому данный способ подавления вывода является более общим, чем переписывание каждого создающего вывод кода. В данном случае его действие проявляется в перехвате (и последующем подавлении) сообщений, выводимых во время обхода, совершаемого `walker.run('.')`. В действительности они выводятся, но не на экран, а в результирующую строку вызова `redirect`:

```
>>> output[:60]
```

```
'. ...\0121 => .\\autoexec.bat\0122 => .\\cleanall.csh\0123 => .\\echoEnv'

>>> import string
>>> len(output), len(string.split(output, '\n'))          # байты, строки
(67609, 1592)

>>> walker.matches
['.\\package.csh', '.\\README-PP2E.txt', '.\\readme-old-pp1E.txt',
'.\\PyTools\\cleanpyc.py', '.\\PyTools\\fixeoln_all.py',
'.\\System\\Processes\\output.txt',
'.\\Internet\\Cgi-Web\\fixcgi.py']
```

Поскольку `redirect` сохраняет выводимый текст в строке, для функций, создающих много вывода, его использование может оказаться менее удачным, чем молчаливых версий `CollectVisitor`. Здесь, например, в строку оперативной памяти было помещено 67 609 байт вывода (смотрите результаты вызова `len`); для некоторых приложений создание такого буфера может оказаться существенным.

Вообще говоря, такая переадресация `sys.stdout` в фиктивные объекты является простым способом отключения вывода (и эквивалентна существующему в Unix понятию переадресации вывода в файл */dev/null*, который выбрасывает все, что в него выводится). Например, мы снова прибегнем к этому трюку в контексте сценариев для Интернета, выполняемых на сервере, чтобы помешать утилитам выводить сообщения о состоянии в выходные потоки генерируемых веб-страниц.[1]

Корректировщики, использующие «обходчиков»

Замечание: если написан и отлажен класс, который умеет делать нечто полезное, например обходить деревья каталогов, легко применить его во всех библиотеках системных утилит. В этом, собственно, весь смысл повторного использования кода. Например, написав классы «обходчиков», представленные в предыдущих разделах, я вскоре переработал оба сценария обхода каталогов *fixnames_all.py* и *fixeoln_all.py*, приведенные в примерах 5.6 и 5.4, соответственно, применив в них модуль visitor вместо собственной логики обхода деревьев (оба они первоначально использовали `find.find`). В примере 5.17 первоначальная функция `convertLines` (исправляющая концы строк в одном файле) объединена с классом, обходящим деревья, из visitor, что дало альтернативную реализацию конвертера концов строк для деревьев каталогов.

Пример 5.17. PP2E\PyTools\visitor_fixeoln.py

```
#########################################################################################
# Использование: "python visitor_fixeoln.py todos|tounix". Переработка fixeoln_all.py
# как подкласса visitor: в этом варианте для составления списка имен используется
# os.path.walk, а не find.find; ограниченно, но быстро: if os.path.splitext(fname)[1] in patts:
#########################################################################################

import visitor, sys, fnmatch, os
from fixeoln_dir import patts
from fixeoln_one import convertEndlines

class EolnFixer(visitor.FileVisitor):
```

[1] Нетерпеливые могут посмотреть на `commonhtml.runsilent` в системе PyMailCgi, представленной в главе 13 «Более крупные примеры сайтов, часть 1». Эта вариация на тему `redirect.redirect`, в которой выводимый текст выбрасывается (а не сохраняется в строке), возвращает значение, возвращаемое вызываемой функцией (а не выводимую ею строку), и позволяет исключительным ситуациям проскакивать оператор `try/finally` (вместо перехвата и сообщения посредством `try/ except`). Однако это все равно действующая переадресация.

```
    def visitfile(self, fullname):                  # поиск базового имени
        basename = os.path.basename(fullname)       # чтобы получить тот же результат
        for patt in patts:                          # меньше посещений
            if fnmatch.fnmatch(basename, patt):
                convertEndlines(self.context, fullname)
                self.fcount = self.fcount + 1       # можно вставить break,
                                                    # но результаты будут разными
if __name__ == '__main__':
    walker = EolnFixer(sys.argv[1])
    walker.run()
    print 'Files matched (converted or not):', walker.fcount
```

Как было показано в главе 2, встроенный модуль `fnmatch` выполняет поиск имен файлов в стиле оболочки Unix; в данном сценарии он используется для поиска имен, соответствующих шаблонам из прежней версии (искать расширения имен файлов за «.» проще, но так мы используем инструмент более общего назначения):

```
C:\temp\examples>python %X%/PyTools/visitor_fixeoln.py tounix
. ...
Changing .\echoEnvironment.pyw
Changing .\Launcher.py
Changing .\Launch_PyGadgets.py
Changing .\Launch_PyDemos.pyw
...здесь удалены строки...
Changing .\PyTools\visitor_find.py
Changing .\PyTools\visitor_fixnames.py
Changing .\PyTools\visitor_find_quiet2.py
Changing .\PyTools\visitor_find_quiet1.py
Changing .\PyTools\fixeoln_one.doc.txt
Files matched (converted or not): 1065

C:\temp\examples>python %X%/PyTools/visitor_fixeoln.py tounix
...здесь удалены строки...
.\Extend\Swig\Shadow ...
.\ ...
.\EmbExt\Exports ...
.\EmbExt\Exports\ClassAndMod ...
.\EmbExt\Regist ...
.\PyTools ...
Files matched (converted or not): 1065
```

Если выполнить этот сценарий и первоначальный *fixeoln_all.py* для дерева примеров книги, то можно заметить, что данная версия посещает на два найденных файла меньше. Это просто отражает то обстоятельство, что `fixeoln_all` согласно своим шаблонам находит и пропускает два имени каталога в результате `find.find` (оба называются «Output»). Во всех остальных отношениях эта версия работает так же, даже когда может поступить лучше – если добавить оператор break после вызова `convertEndlines`, то не будет посещения файлов, избыточно появляющихся в списках результатов поиска оригинала.

Первая команда выполняется на моем компьютере примерно за шесть секунд, а вторая примерно за четыре (не нужно преобразовывать файлы). Это быстрее, чем восемь и шесть секунд для первоначальной версии, использующей `find.find`, но у них разный объем вывода, а измерение производительности обычно значительно более тонкая вещь, чем представляется. Большая часть времени, по-видимому, в действительности расходуется на прокрутку текста в консоли, а не на обработку каталогов. Поскольку для своих целей обе версии достаточно быстры, более точное измерение производительности оставляется в качестве упражнения.

В сценарии примера 5.18 объединены первоначальная функция convertOne (переименовывающая один файл или каталог) и класс, обходящий деревья, из visitor, что дает исправление имен в верхнем регистре для целого дерева каталогов. Обратите внимание на то, что переопределяются методы для обработки как файлов, так и каталогов, поскольку переименовываться должны и те и другие.

Пример 5.18. PP2E\PyTools\visitor_fixnames.py

```
################################################################################
# переработка корректировщика имен fixnames_all.py с помощью класса Visitor
# примечание: "from fixnames_all import convertOne" не помогает на верхнем уровне
# класса fixnames, так как предполагается, что это метод, вызываемый
# с дополнительным аргументом self (исключение);
################################################################################
from visitor import FileVisitor

class FixnamesVisitor(FileVisitor):
    """
    поиск имен в верхнем регистре в startDir и ниже
    """
    import fixnames_all
    def __init__(self, listonly=0):
        FileVisitor.__init__(self, listonly=listonly)
        self.ccount = 0
    def rename(self, pathname):
        if not self.listonly:
            convertflag = self.fixnames_all.convertOne(pathname)
            self.ccount = self.ccount + convertflag
    def visitdir(self, dirname):
        FileVisitor.visitdir(self, dirname)
        self.rename(dirname)
    def visitfile(self, filename):
        FileVisitor.visitfile(self, filename)
        self.rename(filename)

if __name__ == '__main__':
    walker = FixnamesVisitor()
    walker.run()
    allnames = walker.fcount + walker.dcount
    print 'Converted %d files, visited %d' % (walker.ccount, allnames)
```

Эта версия выполняется подобно оригинальной версии fixnames_all, основанной на find.find, но обрабатывает еще одно имя (корневой каталог на верхнем уровне) и не делает задержки в начале, вызываемой составлением списка имен файлов – мы снова используем os.path.walk вместо find.find. Она также похожа на первоначальную версию этого сценария, использующую os.path.walk, но основана на иерархии классов, а не на простых функциях обратного вызова :

```
C:\temp\examples>python %X%/PyTools/visitor_fixnames.py
...здесь удалены строки...
303 => .\__init__.py
304 => .\__init__.pyc
305 => .\Ai\ExpertSystem\holmes.tar
306 => .\Ai\ExpertSystem\TODO
Convert dir=.\Ai\ExpertSystem file=TODO? (y|Y)
307 => .\Ai\ExpertSystem\__init__.py
308 => .\Ai\ExpertSystem\holmes\cnv
309 => .\Ai\ExpertSystem\holmes\README.1ST
```

```
Convert dir=.\Ai\ExpertSystem\holmes file=README.1ST? (y|Y)
...здесь удалены строки...
1353 => .\PyTools\visitor_find.pyc
1354 => .\PyTools\visitor_find_quiet1.py
1355 => .\PyTools\fixeoln_one.doc.txt
Converted 1 files, visited 1474
```

Оба этих корректирующих сценария работают примерно так же, как оригиналы, но поскольку логика обхода каталогов находится в одном файле (*visitor.py*), ее нужно отладить только один раз. Более того, усовершенствования, производимые в этом файле, будут автоматически наследоваться всеми инструментами обработки каталогов, производными от его классов. Даже при написании сценариев системного уровня повторное использование кода и уменьшение избыточности в конечном счете окупаются.

Корректировка прав доступа к файлам в деревьях

В случае если предыдущих разделов, посвященных клиентам модуля обхода, оказалось недостаточно, чтобы убедить вас в мощи многократного использования кода, под конец проекта этой книги появилось еще одно ее свидетельство. Оказывается, при копировании файлов с CD в Windows путем их перетаскивания создаются копии с атрибутом *только для чтения*. Это не самое идеальное свойство для каталога примеров на прилагаемом к книге CD – дерево каталогов нужно скопировать на жесткий диск, чтобы иметь возможность экспериментировать, изменяя программы (естественно, файлы на CD невозможно модифицировать). Но при копировании путем перетаскивания можно получить дерево, содержащее больше 1000 файлов, доступных только для чтения.

Поскольку перетаскивание является, вероятно, самым распространенным способом копирования с CD под Windows, мне понадобился переносимый и простой в использовании способ сбросить установку только для чтения. Просить читателей вручную сделать все эти файлы доступными для записи было бы, по меньшей мере, невежливо. Написание полномасштабной системы для инсталляции, похоже, было бы стрельбой из пушки по воробьям. Обеспечение различных способов исправления для каждой платформы удваивает, если не утраивает, сложность этой задачи.

Гораздо лучше использовать сценарий Python из примера 5.19, который можно выполнить в корне скопированного каталога примеров, чтобы исправить установку только для чтения, совершенную в результате операции перетаскивания. Он снова специализирует обход, реализованный классом `FileVisitor`, – на этот раз так, чтобы для каждого обойденного файла и каталога выполнялся вызов `os.chmod`.

Пример 5.19. PP2E\PyTools\fixreadonly-all.py

```
#!/usr/bin/env python
#########################################################################################
# Использование: python PyTools\fixreadonly-all.py. Запустите этот сценарий в каталоге
# верхнего уровня примеров после копирования всех примеров с CD-ROM книги, чтобы сделать
# все файлы снова доступными для записи – по умолчанию копирование файлов с CD в Windows
# с помощью перетаскивания (по крайней мере) создает их на жестком диске как доступные
# только для чтения; этот сценарий обходит все дерево каталогов начиная с того,
# в котором он запущен, и ниже (все подкаталоги);
#########################################################################################

import os, string
from PP2E.PyTools.visitor import FileVisitor          # оболочка os.path.walk
listonly = 0
```

```
class FixReadOnly(FileVisitor):
    def __init__(self, listonly=0):
        FileVisitor.__init__(self, listonly=listonly)
    def visitDir(self, dname):
        FileVisitor.visitfile(self, dname)
        if self.listonly:
            return
        os.chmod(dname, 0777)
    def visitfile(self, fname):
        FileVisitor.visitfile(self, fname)
        if self.listonly:
            return
        os.chmod(fname, 0777)

if __name__ == '__main__':
    # не выполнять автоматически при щелчке
    go = raw_input('This script makes all files writeable; continue?')
    if go != 'y':
        raw_input('Canceled - hit enter key')
    else:
        walker = FixReadOnly(listonly)
        walker.run()
        print 'Visited %d files and %d dirs' % (walker.fcount, walker.dcount)
```

Как было показано в главе 2, встроенный вызов `os.chmod` изменяет установку прав доступа для внешнего файла (в данном случае на 0777 – глобальное право чтения, записи и выполнения). Поскольку операции `os.chmod` и `FileVisitor` переносимы, тот же самый сценарий установит права доступа для целого дерева как в Windows, так и на Unix-подобных платформах. Обратите внимание, что сценарий при запуске спрашивает, действительно ли вы хотите продолжить, на случай, если кто-либо случайно щелкнет по имени файла в GUI менеджера файлов. Заметьте также, что прежде чем выполнять этот сценарий, делающий файлы доступными для записи, нужно установить Python; это справедливое предположение относительно пользователей, которые собираются изменять сценарии Python.

```
C:\temp\examples>python PyTools\fixreadonly-all.py
This script makes all files writeable; continue?y
. ...
1 => .\autoexec.bat
2 => .\cleanall.csh
3 => .\echoEnvironment.pyw
...здесь удалены строки...
1352 => .\PyTools\visitor_find.pyc
1353 => .\PyTools\visitor_find_quiet1.py
1354 => .\PyTools\fixeoln_one.doc.txt
Visited 1354 files and 119 dirs
```

Копирование деревьев каталогов

В следующих трех разделах, завершающих главу, исследуется ряд дополнительных утилит для обработки каталогов (также называемых «папками») вашего компьютера с использованием Python. В них представлены сценарии для *копирования*, *удаления* и *сравнения* каталогов, демонстрирующие системные инструменты в действии. Все они были порождены необходимостью, в целом переносимы между всеми платформами Python и попутно иллюстрируют концепции разработки программ на Python.

Некоторые из этих сценариев делают нечто слишком исключительное для классов модуля `visitor`, которые применялись в начальных разделах этой главы, и потому требуют особых решений (например, нельзя удалять каталоги, которые мы собираемся обходить). Для большинства из них существуют и специфические для платформ эквиваленты (например, копирование перетаскиванием), но приведенные здесь утилиты на Python являются переносимыми, легко настраиваемыми, могут вызываться из других сценариев и очень быстро работают.

Сценарий Python для копирования дерева

Мой пишущий CD иногда делает странные вещи. Копии файлов с необычными именами могут быть совершенно испорчены на CD несмотря на то, что остальные файлы оказываются целехонькими. При этом необязательно идет насмарку вся работа: если в большой резервной копии на CD испорчено лишь несколько файлов, я всегда могу поштучно скопировать вызвавшие трудности файлы на дискеты. К несчастью, копирование в Windows путем перетаскивания нехорошо поступает с такими CD: копирование прерывается, как только будет обнаружен первый такой испорченный файл. Получить удается столько файлов, сколько было скопировано к моменту возникновения ошибки, и не больше того.

Возможно, в Windows существует какая-нибудь волшебная настройка, позволяющая справиться с этой особенностью, но я перестал ее разыскивать, когда понял, что проще написать копировщик на Python. Сценарий *cpall.py* в примере 5.20 предоставляет один из возможных способов сделать это. С его помощью я могу управлять теми действиями, которые выполняются при обнаружении плохого файла – например, пропустить файл с помощью обработчика исключительной ситуации в Python. Кроме того, это средство работает и на других платформах с тем же интерфейсом и таким же результатом. По крайней мере, мне кажется, что потратить несколько минут и написать переносимый и многократно используемый сценарий Python для решения некоторой задачи более выгодно, чем искать решения, работающие только на одной платформе (если они вообще есть).

Пример 5.20. PP2E\System\Filetools\cpall.py

```
########################################################################################
# Использование: "python cpall.py dir1 dir2". Рекурсивное копирование
# дерева каталогов. Действует подобно команде "cp -r dirFrom/* dirTo" в Unix,
# предполагая, что dirFrom и dirTo являются каталогами. Написан для обхода
# фатальных ошибок при копировании перетаскиванием в Windows (когда первый
# встреченный плохой файл сразу прекращает всю операцию копирования),
# но позволяет также настраивать операции копирования. В Unix могут
# потребоваться дополнительные действия – пропуск ссылок, fifo и т. д.
########################################################################################

import os, sys
verbose = 0
dcount = fcount = 0
maxfileload = 100000
blksize = 1024 * 8

def cpfile(pathFrom, pathTo, maxfileload=maxfileload):
    """
    копировать файл pathFrom в pathTo, байт в байт
    """
    if os.path.getsize(pathFrom) <= maxfileload:
        bytesFrom = open(pathFrom, 'rb').read()      # читать маленький файл целиком
```

```
            open(pathTo, 'wb').write(bytesFrom)          # в Windows требуется режим b
        else:
            fileFrom = open(pathFrom, 'rb')              # читать большие файлы по кускам
            fileTo   = open(pathTo,   'wb')              # здесь тоже нужен режим b
            while 1:
                bytesFrom = fileFrom.read(blksize)       # получить один блок, в конце меньше
                if not bytesFrom: break                  # пусто после последнего куска
                fileTo.write(bytesFrom)

    def cpall(dirFrom, dirTo):
        """
        копировать содержимое dirFrom и ниже в dirTo
        """
        global dcount, fcount
        for file in os.listdir(dirFrom):                 # здесь для файлов/каталогов
            pathFrom = os.path.join(dirFrom, file)
            pathTo   = os.path.join(dirTo,   file)       # получить оба пути
            if not os.path.isdir(pathFrom):              # копировать простые файлы
                try:
                    if verbose > 1: print 'copying', pathFrom, 'to', pathTo
                    cpfile(pathFrom, pathTo)
                    fcount = fcount+1
                except:
                    print 'Error copying', pathFrom, to, pathTo, '--skipped'
                    print sys.exc_type, sys.exc_value
            else:
                if verbose: print 'copying dir', pathFrom, 'to', pathTo
                try:
                    os.mkdir(pathTo)                     # создать новый подкаталог
                    cpall(pathFrom, pathTo)              # рекурсия в подкаталоги
                    dcount = dcount+1
                except:
                    print 'Error creating', pathTo, '--skipped'
                    print sys.exc_type, sys.exc_value

    def getargs():
        try:
            dirFrom, dirTo = sys.argv[1:]
        except:
            print 'Use: cpall.py dirFrom dirTo'
        else:
            if not os.path.isdir(dirFrom):
                print 'Error: dirFrom is not a directory'
            elif not os.path.exists(dirTo):
                os.mkdir(dirTo)
                print 'Note: dirTo was created'
                return (dirFrom, dirTo)

            else:
                print 'Warning: dirTo already exists'
                if dirFrom == dirTo or (hasattr(os.path, 'samefile') and
                                        os.path.samefile(dirFrom, dirTo)):
                    print 'Error: dirFrom same as dirTo'
                else:
                    return (dirFrom, dirTo)

    if __name__ == '__main__':
        import time
        dirstuple = getargs()
```

```
    if dirstuple:
        print 'Copying...'
        start = time.time()
        apply(cpall, dirstuple)
        print 'Copied', fcount, 'files,', dcount, 'directories',
        print 'in', time.time() - start, 'seconds'
```

В этом сценарии реализована собственная логика рекурсивного обхода дерева, в ходе которой запоминаются пути каталогов источника и приемника. На каждом уровне копируются простые файлы, создаются каталоги в целевом пути и происходит рекурсия в подкаталоги с расширением путей «из» и «в» на один уровень. Можно запрограммировать эту задачу и другими способами (например, в других вариантах `cpall` на прилагаемом CD рабочий каталог изменяется в процессе работы с помощью вызовов `os.chdir`), но на практике расширение каталогов при спуске действует хорошо.

Обратите внимание на повторно используемую в этом сценарии функцию `cpfile` — на тот случай, если потребуется копировать файлы размером в гигабайты, он по размеру файла решает, читать ли файл целиком или по кускам (вспомните, что метод файла `read` без аргументов в действительности загружает весь файл в строку, находящуюся в памяти). Заметьте также, что сценарий при необходимости создает целевой каталог, но при начале копирования считает, что он пуст; удалите целевой каталог перед копированием нового дерева с тем же именем (подробнее об этом в следующем разделе).

Вот пример копирования большого дерева примеров книги под Windows; при запуске процесса укажите имена исходного и целевого каталогов и выполните команду оболочки *rm* (или аналогичную для соответствующей платформы), чтобы сначала удалить целевой каталог:

```
C:\temp>rm -rf cpexamples

C:\temp>python %X%\system\filetools\cpall.py examples cpexamples
Note: dirTo was created
Copying...
Copied 1356 files, 118 directories in 2.41999995708 seconds

C:\temp>fc /B examples\System\Filetools\cpall.py
             cpexamples\System\Filetools\cpall.py
Comparing files examples\System\Filetools\cpall.py and
cpexamples\System\Filetools\cpall.py
FC: no differences encountered
```

В этом прогоне дерево из 1356 файлов и 118 каталогов было скопировано за 2,4 секунды на моем 650-мегагерцовом переносном компьютере под Windows 98 (для получения системного времени можно использовать встроенный вызов `time.time`). Если на машине открыты программы типа MS Word, выполнение происходит несколько медленнее, а на вашей машине может оказаться как быстрее, так и медленнее. Во всяком случае это не хуже, чем самые лучшие результаты хронометража при перетаскивании, полученные мной под Windows.

Каким же образом этот сценарий справляется с плохими файлами на резервных CD? Секрет в перехвате и игнорировании файловых исключительных ситуаций и продолжении обхода. Чтобы скопировать с CD все хорошие файлы, я просто выполняю командную строку такого вида:

```
C:\temp>python %X%\system\filetools\cpall_visitor.py g:\PP2ndEd\examples\PP2E cpexamples
```

Поскольку обращение к CD на моей машине под Windows происходит как к «G:», это эквивалент в виде командной строки для копирования путем перетаскивания элемента, находящегося в папке CD верхнего уровня, за исключением того, что сценарий Ру-

thon восстанавливается после ошибок на CD и получает остальные файлы. Вообще говоря, можно передать в cpall любой абсолютный путь к каталогу на вашей машине, даже такому, который означает устройство типа CD. Для выполнения сценария в Linux можно обратиться к приводу CD, указав каталог типа */dev/cdrom*.

Переработка копировщика с использованием класса, основанного на visitor

Впервые создавая только что обсуждавшийся сценарий cpall, я не нашел способа применить в нем иерархию классов visitor, с которой мы познакомились ранее, – требовалось параллельно обходить *два* каталога (оригинала и копии), а visitor основан на обходе одного дерева с помощью os.path.walk. Не было видно простого способа слежения за местонахождением сценария в каталоге копии.

В конечном счете я сообразил, что следить за этим вообще не нужно. Вместо этого сценарий, приведенный в примере 5.21, просто заменяет строку с путем каталога «из» строкой с путем каталога «в» перед всеми именами каталогов и файлов, передаваемых из os.path.walk. В результате замены строк получаются пути, в которые должны копироваться исходные файлы и каталоги.

Пример 5.21. PP2E\System\Filetools\cpall_visitor.py

```
################################################################################
# Использование: "python cpall_visitor.py fromDir toDir" cpall, но с классами
# visitor и os.path.walk; хитрость в замене строк fromDir на toDir перед всеми
# именами, передаваемыми walk; предполагается первоначальное отсутствие toDir;
################################################################################
import os
from PP2E.PyTools.visitor import FileVisitor
from cpall import cpfile, getargs
verbose = 1
class CpallVisitor(FileVisitor):
    def __init__(self, fromDir, toDir):
        self.fromDirLen = len(fromDir) + 1
        self.toDir      = toDir
        FileVisitor.__init__(self)
    def visitdir(self, dirpath):
        toPath = os.path.join(self.toDir, dirpath[self.fromDirLen:])
        if verbose: print 'd', dirpath, '=>', toPath
        os.mkdir(toPath)
        self.dcount = self.dcount + 1
    def visitfile(self, filepath):
        toPath = os.path.join(self.toDir, filepath[self.fromDirLen:])
        if verbose: print 'f', filepath, '=>', toPath
        cpfile(filepath, toPath)
        self.fcount = self.fcount + 1

if __name__ == '__main__':
    import sys, time
    fromDir, toDir = sys.argv[1:3]
    if len(sys.argv) > 3: verbose = 0
    print 'Copying...'
    start = time.time()
    walker = CpallVisitor(fromDir, toDir)
    walker.run(startDir=fromDir)
    print 'Copied', walker.fcount, 'files,', walker.dcount, 'directories',
    print 'in', time.time() - start, 'seconds'
```

Эта версия выполняет примерно ту же задачу, что и первоначальная, но для простоты кода делает несколько допущений: каталог «в» предполагается изначально несуществующим, а возникающие исключительные ситуации не игнорируются. Вот снова образец копирования дерева примеров книги под Windows:

```
C:\temp>rm -rf cpexamples

C:\temp>python %X%\system\filetools\cpall_visitor.py  examples cpexamples -quiet
Copying...
Copied 1356 files, 119 directories in 2.09000003338 seconds

C:\temp>fc /B examples\System\Filetools\cpall.py cpexamples\System\Filetools\cpall.py
Comparing files examples\System\Filetools\cpall.py and
cpexamples\System\Filetools\cpall.py
FC: no differences encountered
```

Несмотря на дополнительную работу со строками, эта версия выполняется столь же быстро, как исходная. Для целей трассировки в этой версии также выводятся во время обхода все пути копирования «из» и «в», если не указать в командной строке третий аргумент или не установить переменную сценария `verbose` равной 0:

```
C:\temp>python %X%\system\filetools\cpall_visitor.py examples cpexamples
Copying...
d examples => cpexamples\
f examples\autoexec.bat => cpexamples\autoexec.bat
f examples\cleanall.csh => cpexamples\cleanall.csh
...здесь удалены строки...
d examples\System => cpexamples\System
f examples\System\System.txt => cpexamples\System\System.txt
f examples\System\more.py => cpexamples\System\more.py
f examples\System\reader.py => cpexamples\System\reader.py
...здесь удалены строки...
Copied 1356 files, 119 directories in 2.31000006199 seconds
```

Удаление деревьев каталогов

Оба сценария для копирования, приведенные в последнем разделе, работают как задумано, но они не равнодушны к уже существующим деревьям каталогов. Это означает, что они неявно полагают, что целевой каталог пуст или вообще отсутствует; в противном случае происходит отказ. Считается, что сначала вы каким-то образом удаляете целевой каталог со своей машины. Для моих задач такое допущение было разумным.

Копировщики можно изменить так, чтобы они могли работать и при существовании каталогов «в» (например, игнорировать исключительные ситуации, генерируемые `os.mkdir`), но я предпочитаю начинать с нуля при копировании деревьев: никогда не знаешь, какой старый мусор лежит в целевом каталоге. Поэтому перед тестированием приведенных сценариев копирования я следил за тем, чтобы выполнить команду `rm -rf cpexamples` и рекурсивно удалить все дерево каталогов *cpexamples*, прежде чем копировать новое дерево с таким же именем.

К сожалению, команда *rm* для очистки целевого каталога в действительности является утилитой Unix, которую я установил на своем PC из коммерческого пакета, и на вашем компьютере ее, возможно, нет. Существуют другие способы удаления деревьев каталогов в зависимости от платформы (например, удалить значок папки в GUI проводника Windows), но почему не сделать это сразу на Python для всех платформ? В примере 5.22 удаляются все файлы и каталоги, начиная с переданного в качестве аргумента каталога и ниже. Поскольку логика работы оформлена в виде функции,

она представляет собой *импортируемую* утилиту, которую можно выполнять из других сценариев. Благодаря использованию чистого кода Python это решение для удаления деревьев работает на разных платформах.

Пример 5.22. PP2E\System\Filetools\rmall.py

```
#!/usr/bin/python
###############################################################
# Использование: "python rmall.py directoryPath directoryPath..."
# Рекурсивное удаление дерева каталогов : удаляет все файлы и каталоги из directoryPaths
# и ниже; спускается в подкаталоги и удаляет родительский каталог последним, т. к. os.rmdir
# требует, чтобы каталог был пуст; аналогично "rm -rf directoryPath" в Unix
###############################################################

import sys, os
fcount = dcount = 0

def rmall(dirPath):                             # удалить dirPath и ниже
    global fcount, dcount
    namesHere = os.listdir(dirPath)
    for name in namesHere:                      # сначала удалить все содержимое
        path = os.path.join(dirPath, name)
        if not os.path.isdir(path):             # удалить простые файлы
            os.remove(path)
            fcount = fcount + 1
        else:                                   # рекурсия для удаления подкаталогов
            rmall(path)
    os.rmdir(dirPath)                           # удалить пустой теперь dirPath
    dcount = dcount + 1

if __name__ == '__main__':
    import time
    start = time.time()
    for dname in sys.argv[1:]: rmall(dname)
    tottime = time.time() - start
    print 'Removed %d files and %d dirs in %s secs' % (fcount, dcount, tottime)
```

Большим преимуществом написания таких средств на Python является то, что у них один и тот же интерфейс командной строки на всех машинах, где установлен Python. Если на вашей машине Windows, Unix или Macintosh отсутствует команда типа `rm -rf`, просто запускайте вместо нее сценарий `rmall`:

```
C:\temp>python %X%\System\Filetools\cpall.py examples cpexamples
Note: dirTo was created
Copying...
Copied 1379 files, 121 directories in 2.68999993801 seconds

C:\temp>python %X%\System\Filetools\rmall.py cpexamples
Removed 1379 files and 122 dirs in 0.549999952316 secs

C:\temp>ls cpexamples
ls: File or directory "cpexamples" is not found
```

Этот сценарий обходит и удаляет дерево из 1379 файлов и 122 каталогов примерно за полсекунды – весьма впечатляюще для некомпилирующего языка программирования и примерно соответствует коммерческой программе `rm –rf`, которую я купил и установил на своем PC.

Одна тонкость: сценарий должен следить за тем, чтобы содержимое каталога удалялось *прежде* самого каталога – вызов `os.rmdir` требует, чтобы удаляемый каталог был

пуст (и возбуждает исключительную ситуацию, если это не так). По этой причине рекурсивные вызовы для подкаталогов должны происходить перед вызовом `os.mkdir`. Специалист по информатике распознал бы здесь обход «сначала вглубь», поскольку родительские каталоги мы обрабатываем после дочерних. Это также исключает использование при обходе `os.path.walk`: необходимо *возвратиться* в родительский каталог, чтобы удалить его после посещения его потомков.

Для иллюстрации запустим в интерактивном режиме вызовы `os.remove` и `os.rmdir` с каталогом *cpexample*, содержащим файлы или вложенные каталоги:

```
>>> os.path.isdir('cpexamples')
1
>>> os.remove('cpexamples')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OSError: [Errno 2] No such file or directory: 'cpexamples'
>>> os.rmdir('cpexamples')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OSError: [Errno 13] Permission denied: 'cpexamples'
```

Оба вызова дают ошибку, если каталог не пуст. Теперь удалим содержимое *cpexamples* из другого окна и попытаемся снова:

```
>>> os.path.isdir('cpexamples')
1
>>> os.remove('cpexamples')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OSError: [Errno 2] No such file or directory: 'cpexamples'
>>> os.rmdir('cpexamples')
>>> os.path.exists('cpexamples')
0
```

Вызов `os.remove` все равно дает ошибку – он предназначен только для удаления элементов, не являющихся каталогами, – но `os.rmdir` теперь работает, поскольку каталог пуст. Вывод из этого такой, что при обходе дерева для удаления каталоги в целом должны удаляться «на обратном пути».

Переработка удаления для придания общности

В существующем виде сценарий `rmall` обрабатывает только имена каталогов и дает ошибку при получении имени обычного файла, но его можно очень просто обобщить так, чтобы снять это ограничение. Версия в примере 5.23 принимает в командной строке любой список имен файлов и каталогов, удаляет обычные файлы и рекурсивно удаляет каталоги.

Пример 5.23. PP2E\System\Filetools\rmall2.py

```
#!/usr/bin/python
###################################################################
# Использование: "python rmall2.py fileOrDirPath fileOrDirPath..."
# Похож на rmall.py, но допускает файлы в командной строке
###################################################################

import sys, os
fcount = dcount = 0

def rmone(pathName):
```

```
    global fcount, dcount
    if not os.path.isdir(pathName):        # удалить обычные файлы
        os.remove(pathName)
        fcount = fcount + 1
    else:                                  # рекурсия для удаления содержимого
        for name in os.listdir(pathName):
            rmone(os.path.join(pathName, name))
        os.rmdir(pathName)                 # удалить пустой теперь dirPath
        dcount = dcount + 1

if __name__ == '__main__':
    import time
    start = time.time()
    for name in sys.argv[1:]: rmone(name)
    tottime = time.time() - start
    print 'Removed %d files and %d dirs in %s secs' % (fcount, dcount, tottime)
```

Эта сокращенная версия делает то же, что исходная, и так же быстро:

```
C:\temp>python %X%\System\Filetools\cpall.py examples cpexamples
Note: dirTo was created
Copying...
Copied 1379 files, 121 directories in 2.52999997139 seconds

C:\temp>python %X%\System\Filetools\rmall2.py cpexamples
Removed 1379 files and 122 dirs in 0.550000071526 secs

C:\temp>ls cpexamples
ls: File or directory "cpexamples" is not found
```

Но с ее помощью можно удалять обычные файлы:

```
C:\temp>python %X%\System\Filetools\rmall2.py spam.txt eggs.txt
Removed 2 files and 0 dirs in 0.0600000619888 secs

C:\temp>python %X%\System\Filetools\rmall2.py spam.txt eggs.txt cpexamples
Removed 1381 files and 122 dirs in 0.630000042915 secs
```

Как всегда, есть несколько способов сделать это в Python (хотя придется постараться, чтобы найти их все). Заметьте, что в этих сценариях не перехватываются исключительные ситуации: в программах, предназначенных для повального удаления целого дерева каталогов, исключительная ситуация должна, по всей видимости, означать нечто весьма нехорошее. Можно заняться украшательством и организовать поддержку шаблонов имен файлов с помощью встроенного модуля `fnmatch`, но это выходит за пределы задач, которые должны были решаться данными сценариями (по поводу поиска по шаблону смотрите пример 5.17 и *find.py* в главе 2).

Сравнение деревьев каталогов

У инженеров бывают параноидальные наклонности (но я вам этого не говорил). По крайней мере, это относится ко мне. Я полагаю, что все это от многолетнего наблюдения за тем, как происходят ужасные вещи. Например, когда я делаю на CD резервную копию жесткого диска, то чувствую в этом процессе нечто слишком таинственное, чтобы доверяться программе записи CD. Может быть, это неправильно, но я не могу сильно доверять инструментам, которые иногда калечат файлы и по третьим четвергам каждого месяца вызывают крах моей машины с Windows 98. На крайний случай неплохо иметь возможность как можно скорее проверить, что скопированные на CD

данные совпадают с исходными, или хотя бы установить, где находятся различия. Если резервная копия вообще нужна, то когда-нибудь она *действительно* понадобится.

Поскольку доступ к CD с данными осуществляется как к обычным деревьям каталогов, мы еще раз попадаем в сферу действия обходчиков деревьев – для проверки CD с резервной копией нужно сделать обход его каталога верхнего уровня. Мы уже написали общий класс обходчика (модуль `visitor`), но непосредственно не сможем им здесь воспользоваться: необходимо параллельно обойти *два* каталога и исследовать по дороге файлы. Кроме того, обход только одного из двух каталогов не позволит выявить файлы и каталоги, существующие только в другом каталоге. Здесь требуется некоторое более специальное решение.

Поиск расхождений между каталогами

Прежде чем начать писать программу, необходимо выяснить, что значит сравнить два дерева каталогов. Если оба дерева имеют одинаковую структуру ветвей и глубину, проблема сводится к сравнению соответствующих файлов в каждом дереве. Однако, вообще говоря, деревья могут иметь произвольную различную форму, глубину и т. д.

В более общем случае в содержимом каталога одного дерева может быть больше или меньше элементов, чем в соответствующем каталоге другого дерева. Если это различное содержимое представляет собой имена файлов, то нет соответствующих файлов для сравнения; если это имена каталогов, то нет соответствующей ветви, в которую нужно войти. На самом деле единственный способ выявить файлы и каталоги, которые есть в одном дереве, но отсутствуют в другом, – это выявлять различия в каталогах каждого уровня.

Иными словами, алгоритм сравнения деревьев должен будет также попутно выполнять сравнение *каталогов*. Начнем с программирования сравнения имен файлов для одного каталога, показанного в примере 5.24, так как это вложенная и более простая операция.

Пример 5.24. PP2E\System\Filetools\dirdiff.py

```
#!/bin/env python
########################################################################################
# Использование: python dirdiff.py dir1-path dir2-path
# Сравнение двух каталогов для обнаружения файлов, существующих в одном каталоге,
# но не в другом; в данной версии используется функция os.listdir и перечисляются
# различия; обратите внимание, что этот сценарий проверяет только имена файлов,
# а не их содержимое – см. расширение diffall.py, сравнивающее содержимое
# путем сравнения результатов .read();
########################################################################################

import os, sys

def reportdiffs(unique1, unique2, dir1, dir2):
    if not (unique1 or unique2):
        print 'Directory lists are identical'
    else:
        if unique1:
            print 'Files unique to', dir1
            for file in unique1:
                print '...', file
        if unique2:
            print 'Files unique to', dir2
            for file in unique2:
                print '...', file
```

```
def unique(seq1, seq2):
    uniques = []                            # возвратить элементы, присутствующие только в seq1
    for item in seq1:
        if item not in seq2:
            uniques.append(item)
    return uniques

def comparedirs(dir1, dir2):
    print 'Comparing', dir1, 'to', dir2
    files1  = os.listdir(dir1)
    files2  = os.listdir(dir2)
    unique1 = unique(files1, files2)
    unique2 = unique(files2, files1)
    reportdiffs(unique1, unique2, dir1, dir2)
    return not (unique1 or unique2)               # истина, если нет различий

def getargs():
    try:
        dir1, dir2 = sys.argv[1:]                 # 2 аргумента командной строки
    except:
        print 'Usage: dirdiff.py dir1 dir2'
        sys.exit(1)
    else:
        return (dir1, dir2)

if __name__ == '__main__':
    dir1, dir2 = getargs()
    comparedirs(dir1, dir2)
```

Получив списки имен для каждого каталога, этот сценарий просто выбирает уникальные имена в первом каталоге, уникальные имена во втором каталоге и сообщает о найденных уникальных именах как о расхождениях (то есть файлах, имеющихся в одном каталоге, но не в другом). Функция `comparedirs` возвращает истинное значение, если расхождения не были обнаружены, что полезно для обнаружения различий при вызове из других программ.

Запустим этот сценарий с несколькими каталогами; он сообщает о тех найденных различиях, которые представляют уникальные имена в любом из переданных путей каталогов. Обратите внимание на то, что при этом сравниваются только *структуры* путем проверки имен в списках, но не содержимое файлов (последнее будет вскоре добавлено):

```
C:\temp>python %X%\system\filetools\dirdiff.py examples cpexamples
Comparing examples to cpexamples
Directory lists are identical

C:\temp>python %X%\system\filetools\dirdiff.py examples\PyTools cpexamples\PyTools
Comparing examples\PyTools to cpexamples\PyTools
Files unique to examples\PyTools
... visitor.py

C:\temp>python %X%\system\filetools\dirdiff.py  examples\System\Filetools
                                 cpexamples\System\Filetools
Comparing examples\System\Filetools to cpexamples\System\Filetools
Files unique to examples\System\Filetools
... dirdiff2.py
Files unique to cpexamples\System\Filetools
... cpall.py
```

В основе сценария лежит функция `unique`: она осуществляет простую операцию вычитания списков. Вот как она работает, независимо от остальной части кода:

```
>>> L1 = [1, 3, 5, 7, 9]
>>> L2 = [2, 3, 6, 8, 9]
>>> from dirdiff import unique
>>> unique(L1, L2)                  # элементы в L1, но не в L2
[1, 5, 7]
>>> unique(L2, L1)                  # элементы в L2, но не вL1
[2, 6, 8]
```

В этих двух списках общими элементами являются 3 и 9; остальные присутствуют только в одном списке из двух. В применении к каталогам уникальные элементы представляют различия между деревьями, а общие элементы представляют имена файлов или подкаталогов, которые достойны дальнейшего сравнения или обхода. Этот код можно проверить другими способами, некоторые из которых можно найти в версиях `dirdiff` на прилагаемом CD.

Нахождение различий между деревьями

Теперь нам нужен только обходчик дерева, который применяет `dirdiff` на каждом уровне, чтобы отобрать уникальные файлы и каталоги, явно сравнивает содержимое общих файлов и проходит через общие каталоги. Это осуществляет код примера 5.25.

Пример 5.25. PP2E\System\Filetools\diffall.py

```
############################################################################################
# Использование: "python diffall.py dir1 dir2". Рекурсивное сравнение деревьев – сообщить
# о файлах, имеющихся только в одном из dir1 и dir2, сообщить об одноименных файлах в dir1
# и dir2 с разным содержанием и сделать то же для всех одноименных подкаталогов в dir1
# и dir2 и ниже их; сводка различий появляется в конце вывода (дополнительные подробности
# нужно искать под строками "DIFF" и "unique" в переадресованном выводе);
############################################################################################
import os, dirdiff
def intersect(seq1, seq2):
    commons = []                # общие элементы seq1 и seq2
    for item in seq1:
        if item in seq2:
            commons.append(item)
    return commons

def comparedirs(dir1, dir2, diffs, verbose=0):
    # сравнить списки имен файлов
    print '-'*20
    if not dirdiff.comparedirs(dir1, dir2):
        diffs.append('unique files at %s - %s' % (dir1, dir2))

    print 'Comparing contents'
    files1 = os.listdir(dir1)
    files2 = os.listdir(dir2)
    common = intersect(files1, files2)

    # сравнить содержимое общих файлов
    for file in common:
        path1 = os.path.join(dir1, file)
        path2 = os.path.join(dir2, file)
        if os.path.isfile(path1) and os.path.isfile(path2):
            bytes1 = open(path1, 'rb').read()
            bytes2 = open(path2, 'rb').read()
```

```
            if bytes1 == bytes2:
                if verbose: print file, 'matches'
            else:
                diffs.append('files differ at %s - %s' % (path1, path2))
                print file, 'DIFFERS'

    # рекурсия для сравнения общих каталогов
    for file in common:
        path1 = os.path.join(dir1, file)
        path2 = os.path.join(dir2, file)
        if os.path.isdir(path1) and os.path.isdir(path2):
            comparedirs(path1, path2, diffs, verbose)

if __name__ == '__main__':
    dir1, dir2 = dirdiff.getargs()
    mydiffs = []
    comparedirs(dir1, dir2, mydiffs)        # изменяет mydiffs по месту
    print '='*40                            # обход, вывод списка diffs
    if not mydiffs:
        print 'No diffs found.'
    else:
        print 'Diffs found:', len(mydiffs)
        for diff in mydiffs: print '-', diff
```

В каждом каталоге этого дерева данный сценарий просто выполняет утилиту `dirdiff`, чтобы обнаружить уникальные имена, а затем сравнивает общие имена, находящиеся в пересечении перечней каталогов. Так как мы уже изучали средства обхода деревьев, примененные в этом сценарии, перейдем прямо к некоторым примерам выполнения. При обработке идентичных деревьев во время обхода выводятся сообщения о состоянии, а в конце появляется сообщение: «No diffs found» (расхождений нет):

```
C:\temp>python %X%\system\filetools\diffall.py examples cpexamples
--------------------
Comparing examples to cpexamples
Directory lists are identical
Comparing contents
--------------------
Comparing examples\old_Part2 to cpexamples\old_Part2
Directory lists are identical
Comparing contents
--------------------
...здесь удалены строки...
--------------------
Comparing examples\EmbExt\Regist to cpexamples\EmbExt\Regist
Directory lists are identical
Comparing contents
--------------------
Comparing examples\PyTools to cpexamples\PyTools
Directory lists are identical
Comparing contents
========================================
No diffs found.
```

Чтобы посмотреть, как сообщается о расхождениях, нужно их создать. Запустим встречавшийся нам раньше сценарий глобального поиска и замены, чтобы изменить несколько файлов в одном из деревьев (я ведь предупреждал, что глобальная замена может разрушить файлы!):

```
C:\temp\examples>python %X%\PyTools\visitor_replace.py -exec SPAM
```

```
Are you sure?y
...
1355 => .\PyTools\visitor_find_quiet1.py
1356 => .\PyTools\fixeoln_one.doc.txt
Visited 1356 files
Changed 8 files:
.\package.csh
.\README-PP2E.txt
.\readme-old-pp1E.txt
.\temp
.\remp
.\Internet\Cgi-Web\fixcgi.py
.\System\Processes\output.txt
.\PyTools\cleanpyc.py
```

С той же целью удалим несколько файлов, чтобы посмотреть на сообщения об уникальных файлах в каталогах; следующие три команды удаления сделают различными два каталога (последние две команды действуют на один и тот же каталог в разных деревьях):

```
C:\temp>rm cpexamples\PyTools\visitor.py
C:\temp>rm cpexamples\System\Filetools\dirdiff2.py
C:\temp>rm examples\System\Filetools\cpall.py
```

Теперь перезапустим сценарий сравнения, чтобы обнаружить различия, и перенаправим вывод в файл, чтобы облегчить ее просмотр. Ниже приведена лишь часть выходного отчета, в которой сообщается о различиях. При обычном использовании я сначала смотрю на сводку в конце отчета, а затем ищу в тексте отчета строки «DIFF» и «unique», если мне нужна дополнительная информация об отличиях, указанных в сводке:

```
C:\temp>python %X%\system\filetools\diffall.py examples cpexamples > diffs
C:\temp>type diffs
--------------------
Comparing examples to cpexamples
Directory lists are identical
Comparing contents
package.csh DIFFERS
README-PP2E.txt DIFFERS
readme-old-pp1E.txt DIFFERS
temp DIFFERS
remp DIFFERS
--------------------
Comparing examples\old_Part2 to cpexamples\old_Part2
Directory lists are identical
Comparing contents
--------------------
...
--------------------
Comparing examples\Internet\Cgi-Web to cpexamples\Internet\Cgi-Web
Directory lists are identical
Comparing contents
fixcgi.py DIFFERS
--------------------
...
--------------------
Comparing examples\System\Filetools to cpexamples\System\Filetools
Files unique to examples\System\Filetools
```

```
... dirdiff2.py
Files unique to cpexamples\System\Filetools
... cpall.py
Comparing contents
--------------------
...
--------------------
Comparing examples\System\Processes to cpexamples\System\Processes
Directory lists are identical
Comparing contents
output.txt DIFFERS
--------------------
...
--------------------
Comparing examples\PyTools to cpexamples\PyTools
Files unique to examples\PyTools
... visitor.py
Comparing contents
cleanpyc.py DIFFERS
========================================
Diffs found: 10
- files differ at examples\package.csh - cpexamples\package.csh
- files differ at examples\README-PP2E.txt - cpexamples\README-PP2E.txt
- files differ at examples\readme-old-pp1E.txt - cpexamples\readme-old-pp1E.txt
- files differ at examples\temp - cpexamples\temp
- files differ at examples\remp - cpexamples\remp
- files differ at examples\Internet\Cgi-Web\fixcgi.py -
            cpexamples\Internet\Cgi-Web\fixcgi.py
- unique files at examples\System\Filetools -
            cpexamples\System\Filetools
- files differ at examples\System\Processes\output.txt -
            cpexamples\System\Processes\output.txt
- unique files at examples\PyTools - cpexamples\PyTools
- files differ at examples\PyTools\cleanpyc.py - cpexamples\PyTools\cleanpyc.py
```

Я добавил переносы строки и отступы в некоторых строках вывода, чтобы уместить их на странице, но отчет легко понять. Было найдено десять различий – восемь файлов были изменены сценарием замены, а два каталога мы рассогласовали тремя командами удаления *rm*.

Проверка резервных копий на CD

Каким же образом этот сценарий успокаивает паранойю при создании резервных копий на CD? Для тщательной проверки работы моего пишущего привода CD я выполняю команду типа приведенной ниже. С помощью такой команды я могу также найти изменения, произведенные после предыдущего резервного копирования. И снова, поскольку включенный на моей машине CD представляется как «G:», я указываю путь с таким корнем; на Linux используйте корень типа */dev/cdrom*:

```
C:\temp>python %X%\system\filetools\diffall.py
                    examples g:\PP2ndEd\examples\PP2E > exdiffs091500

C:\temp>more exdiffs091500
--------------------
Comparing examples to g:\PP2ndEd\examples\PP2E
Files unique to examples
... .cshrc
Comparing contents
```

```
tounix.py DIFFERS
--------------------
Comparing examples\old_Part2 to g:\PP2ndEd\examples\PP2E\old_Part2
Directory lists are identical
Comparing contents
--------------------
...еще строки
visitor_fixeoln.py DIFFERS
visitor_fixnames.py DIFFERS
========================================
Diffs found: 41
- unique files at examples - g:\PP2ndEd\examples\PP2E
- files differ at examples\tounix.py - g:\PP2ndEd\examples\PP2E\tounix.py
...еще строки
```

CD крутится, сценарий сравнивает, и в конце отчета появляется сводка из 41 различия (в данном случае представляющая изменения в дереве примеров, выполненные после записи последнего CD с резервной копией). Пример полного отчета о различиях находится в файле *exdiffs091500* на прилагаемом CD. Чаще всего с большинством резервных копий моих примеров случается вот что: файлы, начинающиеся с «.», не копируются на CD:

```
C:\temp>python %X%\System\Filetools\diffall.py
                    examples g:\PP2ndEd\examples\PP2E
...
--------------------
Comparing examples\Config to g:\PP2ndEd\examples\PP2E\Config
Files unique to examples\Config
... .cshrc
Comparing contents
========================================
Diffs found: 1
- unique files at examples\Config - g:\PP2ndEd\examples\PP2E\Config
```

А чтобы быть *действительно* уверенным, я выполняю следующую команду глобального сравнения с настоящим каталогом книги, чтобы проверить целиком резервную копию дерева книги на CD:

```
C:\>python %X%\System\Filetools\diffall.py PP2ndEd G:\PP2ndEd
--------------------
Comparing PP2ndEd to G:\PP2ndEd
Files unique to G:\PP2ndEd
... examples.tar.gz
Comparing contents
README.txt DIFFERS
--------------------
...еще строки
--------------------
Comparing PP2ndEd\examples\PP2E\Config to G:\PP2ndEd\examples\PP2E\Config
Files unique to PP2ndEd\examples\PP2E\Config
... .cshrc
Comparing contents
--------------------
...еще строки
--------------------
Comparing PP2ndEd\chapters to G:\PP2ndEd\chapters
Directory lists are identical
Comparing contents
```

```
ch01-intro.doc DIFFERS
ch04-os3.doc DIFFERS
ch05-gui1.doc DIFFERS
ch06-gui2.doc DIFFERS
--------------------
...еще строки
========================================
Diffs found: 11
- unique files at PP2ndEd - G:\PP2ndEd
- files differ at PP2ndEd\README.txt - G:\PP2ndEd\README.txt
...еще строки
```

Данный прогон показывает, что после создания предыдущей копии я изменил файл «readme», четыре файла глав и еще кое-что; если выполнить эту команду сразу после создания резервной копии, то радар `diffall` показывает только уникальный файл *.cshrc*. Такое глобальное сравнение может занять несколько минут: оно осуществляет побайтовое сравнение всех файлов глав и снимков экрана, дерева примеров, образа прилагаемого CD и других, но это точное и полное сравнение. При условии, что дерево этой книги согласно последней проверке содержит примерно 119 Мбайт данных в 7300 файлах и 570 каталогах, какая-либо менее автоматизированная процедура верификации без помощи Python была бы совершенно немыслима.

Наконец, стоит заметить, что этот сценарий только *обнаруживает* расхождения в деревьях, не сообщая никаких подробностей о различиях в отдельных файлах. На самом деле он просто загружает и сравнивает двоичное содержимое соответствующих файлов как одной строки, давая простой результат «да/нет».[1] В тех случаях, когда мне нужны дополнительные сведения о фактических различиях в не совпавших файлах, я либо открываю их в редакторе, либо выполняю команду сравнения файлов на соответствующей платформе (например, *fc* в Windows/DOS, *diff* или *cmp* в Unix и Linux). Этот последний шаг не является переносимым решением, но для решаемых мною задач просто нахождение различий в дереве из 1300 файлов было значительно более важным, чем сообщение в отчете о том, в каких строках различаются эти файлы.

Конечно, поскольку в Python всегда можно вызвать команды оболочки, этот последний шаг можно автоматизировать, порождая при обнаружении различий команду *diff* или *fc* с помощью `os.popen` (или делать это после обхода, сканируя содержащуюся в отчете сводку). Однако поскольку Python отлично обрабатывает файлы и строки, можно сделать еще один шаг и написать на Python эквивалент команд *fc* и *diff*. Поскольку это выходит за рамки задач данного сценария и размеров этой главы, его предоставляется выполнить любознательному читателю.

---

[1] Можно было бы попробовать поступить лучше, открывая текстовые файлы в текстовом режиме, игнорируя различия в терминаторах строк, но не очевидно, что такие различия должны быть просто проигнорированы (что если вызывающей программе нужно узнать, не изменились ли маркеры конца строки?). Можно было бы также поступить умнее и не выполнять загрузку и сравнение файлов, которые отличаются размерами, и читать файлы маленькими кусками, а не все целиком, чтобы минимизировать расход памяти для очень больших файлов (указания по этому поводу можно найти в более ранних примерах, например сценарии `cpall`). Для тех сравнений, которые нужны мне, такие оптимизации излишни.

II

Программирование GUI

Эта часть книги показывает, как применять Python для создания переносимых графических интерфейсов пользователя в основном с помощью стандартной библиотеки Python Tkinter. Следующие главы глубоко освещают эту тему:

- Глава 6 «Графические интерфейсы пользователя». Эта глава очерчивает возможности создания GUI, доступные разработчикам Python, а затем представляет учебный материал, иллюстрирующий базовые понятия кодирования с использованием Tkinter в контексте простых интерфейсов пользователя.
- Глава 7 «Обзор Tkinter, часть 1». Эта глава начинает обзор библиотеки Tkinter – используемый набор графических элементов и сопутствующих инструментов. В первой части обзора рассказывается о наиболее простых инструментах и графических элементах библиотеки: всплывающих окнах, различных видах кнопок и т. д.
- Глава 8 «Обзор Tkinter, часть 2». Здесь продолжается обзор библиотеки, начатый в предыдущей главе. Представлена остальная часть библиотеки графических элементов Tkinter – меню, графические изображения, текст, холст, сетки, события таймера и анимация.
- Глава 9 «Более крупные примеры GUI». Эта глава соединяет идеи из предыдущих глав для реализации набора интерфейсов пользователя. Начинается она с рассмотрения приемов автоматизации GUI и завершается представлением более крупных GUI – часов, текстовых редакторов, программ для рисования, средств просмотра графических изображений и других.

Как и в первой части книги, представленный здесь материал применим в целом ряде областей и будет снова использован в дальнейшем для создания специфических для областей применения интерфейсов в последующих главах книги.

6

Графические интерфейсы пользователя

«Я здесь, я смотрю на тебя, детка»

Для большинства программных систем графический интерфейс пользователя (GUI) стал привычной частью пакета. Даже если акроним «GUI» вам незнаком, вы, вероятно, знакомы с такими элементами, как окна, кнопки и меню, используемыми при работе с программами. В действительности большая часть работы с компьютерами сегодня осуществляется с помощью какого-нибудь графического интерфейса «укажи и щелкни». Программы – от веб-броузера до системных инструментов – стандартно оснащаются компонентами GUI, повышающими гибкость и простоту их использования.

В этой части мы научимся заставлять сценарии Python порождать такие графические интерфейсы, изучая примеры программирования с помощью модуля Tkinter – переносимой библиотеки GUI, стандартно входящей в систему Python. Как будет показано, интерфейсы пользователя легко программируются в сценариях Python благодаря простоте языка и мощи его библиотек GUI. Дополнительным преимуществом является автоматическая переносимость GUI, запрограммированных на Python с использованием Tkinter, на большинство компьютерных систем.

Темы программирования GUI

Поскольку GUI представляют собой большую область, я хочу сказать еще несколько слов об этой части книги. Для облегчения усвоения темы программирования GUI разделены между следующими четырьмя главами этой книги:

- Данная глава начинается с краткого учебника по Tkinter, знакомящего с основами кодирования. Здесь намеренно сохраняется простота интерфейсов, чтобы вы могли овладеть базовыми знаниями, прежде чем перейти к интерфейсам из следующей главы. С другой стороны, в этой главе полностью освещены основы: обработка событий, менеджер геометрии `pack`, использование наследования и композиции в GUI и другое. Как будет показано, ООП (объектно-ориентированное программирование) не является обязательным для Tkinter, но оно делает GUI структурированными и многократно используемыми.
- Главы 7 и 8 представляют обзор набора графических элементов (widgets) модуля Tkinter.[1] Грубо говоря, глава 7 «Обзор Tkinter, часть 1» представляет простые гра-

[1] Термин «набор графических элементов» (widget set) означает объекты, используемые для создания привычных элементов интерфейса «point-and-click» (укажи-и-щелкни) – кнопок, ползунков, полей для ввода и т. д. В Tkinter есть классы Python, соответствующие всем графическим элементам, к которым вы привыкли в графических интерфейсах. Помимо графических элементов в Tkinter есть средства другого рода – планирование событий, ожидание данных в сокетах и т. д.

фические элементы, а глава 8 «Обзор Tkinter, часть 2» охватывает более сложные графические элементы и связанные с ними инструменты. Здесь встречается большинство привычных элементов интерфейсов: ползунки, меню, диалоги, графика и тому подобное. Эти две главы не могут служить полным справочником по Tkinter (который вполне мог бы сам составить большую книгу), но их должно хватить для того, чтобы начать кодировать серьезные GUI на Python. Примеры в этих главах сосредоточены на графических элементах и инструментах Tkinter, но попутно исследуется поддержка повторного использования кода в Python.

- В главе 9 «Более крупные примеры GUI» представлены более сложные примеры, в которых используются техника кодирования и графические элементы, показанные в трех предшествующих главах. Она начинается с изучения техники автоматизации часто встречающихся задач GUI с помощью Python. Хотя Tkinter является полнофункциональной библиотекой, некоторый объем повторно используемого кода Python может сделать его интерфейсы еще более мощными и простыми в использовании. Эта глава завершается показом нескольких законченных программ GUI, в которых реализованы текстовые редакторы, средства просмотра графики, часы и прочее.

Поскольку GUI в действительности являются инструментами, используемыми в разных областях, другие примеры GUI будут появляться на протяжении всей оставшейся части книги. Например, позднее мы увидим GUI для работы с электронной почтой, калькуляторы, средства показа деревьев, броузеры таблиц и т. д. Смотрите в главе 4 по GUI список предварительных ссылок на другие примеры Tkinter в данной книге.

Сразу же хочу отметить одно обстоятельство: большинство GUI являются динамическими и интерактивными интерфейсами, и мне не остается здесь ничего иного, как показать статические снимки экрана, представляющие избранные состояния взаимодействия, реализуемого такими программами. Для большинства примеров это несправедливо. Если вы еще не работаете с примерами, рекомендую самостоятельно запускать примеры GUI в этой и последующей главах.

Кто-нибудь заметил, что «GUI» совпадает с тремя первыми буквами «GUIDO»?

Создатель Python первоначально не собирался создавать инструмент для разработки GUI, но простота использования Python и краткость цикла разработки сделали это одной из главных его функций. С точки зрения реализации GUI в Python являются в действительности примерами расширений C, а расширяемость была одной из главных идей при создании Python. Когда сценарий создает кнопки и меню, он в конечном счете обращается к библиотеке C, а когда сценарий реагирует на пользовательское событие, библиотека C в конечном счете обращается обратно к Python.

Но с практической точки зрения GUI являются важной частью современных систем и идеальной областью применения таких средств, как Python. Как будет показано, простота синтаксиса Python и его объектная ориентированность хорошо сочетаются с моделью GUI – каждый изображаемый на экране графический элемент естественным образом представляется в виде класса Python. Кроме того, краткость цикла разработки в Python позволяет программистам быстро экспериментировать с вариантами расположения элементов и их поведением, что невозможно при обычной технологии разработки. На практике обычно можно за считанные секунды сделать изменения в GUI, созданном на Python, и посмотреть на их результат. На C или C++ это невозможно.

В Windows стандартная установка Python имеет встроенную поддержку Tkinter, поэтому все эти примеры должны работать сразу. Для других систем тоже легко получить Python с поддержкой Tkinter (подробности смотрите в приложении B «Прагматика» и файле верхнего уровня *README-PP2E.txt*). Стоит, однако, несмотря на возможную необходимость дополнительной инсталляции, поэкспериментировать с этим программами, так как это хороший способ изучения как программирования GUI, так и самого Python.

Варианты разработки GUI в Python

Прежде чем погрузиться в Tkinter, рассмотрим перспективные варианты разработки GUI в Python в целом. Поскольку Python показал себя вполне подходящим для работы с GUI, в этой области в последние годы наблюдалась высокая активность. На самом деле, хотя в качестве инструментария GUI в Python чаще всего используется Tkinter, на сегодняшний день есть несколько способов программирования пользовательских интерфейсов в Python. Некоторые являются специфическими для Windows или X Windows,[1] другие представляют собой решения для нескольких платформ, и у всех них есть свои приверженцы и сильные стороны. Чтобы быть справедливым ко всем возможным альтернативам, приведем краткий перечень инструментария GUI, доступного для программистов Python на момент написания этих слов:

Tkinter (поставляется с Python)

Переносимая библиотека GUI с открытым исходным кодом, используемая как стандарт де-факто для разработки GUI в Python. Tkinter облегчает быстрое построение простых GUI и может быть легко дополнен в Python более крупными структурами компонентов. Сценарии Python, использующие Tkinter для построения GUI, выполняются переносимым образом в Windows, X Windows (Unix) и Macintosh, создавая на каждой из этих платформ свойственный для нее внешний вид. Библиотека Tk, на которой основан Tkinter, является стандартом в мире open source в целом и используется также языками сценариев Perl и Tcl.

wxPython (http://wxpython.org)

Интерфейс Python с открытым исходным кодом для wxWindows – переносимая структура классов GUI, первоначально написанная для использования из языка программирования C++. Система wxPython является модулем расширения, служащим оболочкой для классов wxWindows. Эта библиотека, по общему мнению, считается превосходной для создания сложных интерфейсов и сегодня является, вероятно, вторым по популярности инструментальным набором GUI в Python после Tkinter. Когда пишутся эти строки, код wxPython переносим на Windows и Unix-подобные платформы, но не на Macintosh. Используемая в основе библиотека лучше всего поддерживает Windows и GTK (в Unix), но в целом переносима в Windows, Unix-подобные платформы и Macintosh.

JPython (http://www.jpython.org)

Как мы увидим в главе 15 «Более сложные темы Интернета», JPython (известный также как «Jython») является вариантом Python для Java, который дает сценари-

[1] В данной книге «Windows» относится к интерфейсу Microsoft Windows, распространенному на PC, а «X Windows» (правильнее «X Window» – *примеч. перев.*) относится к интерфейсу X11, чаще всего встречающемуся на платформах Unix и Linux. Эти два интерфейса в целом привязаны к платформам Microsoft и Unix, соответственно. Существует возможность выполнения X Windows поверх операционной системы Microsoft и эмуляции Windows в Unix и Linux, но это не распространено широко.

ям Python беспрепятственный доступ к библиотекам классов Java на локальной машине. Благодаря этому библиотеки GUI Java, такие как `swing` и `awt`, дают еще один способ построения GUI в коде Python, выполняемом в системе JPython. Очевидно, такие решения являются специфическими для Java, и их переносимость ограничена переносимостью Java и его библиотек. Новый пакет под названием `jTkinter` предоставляет также перенос Tkinter в JPython с использованием JNI Java; при его установке сценарии Python могут также использовать Tkinter для построения GUI под JPython.

KDE и Qt (http://www.thekompany.com/projects/pykde)
Gnome и GTK (ftp://ftp.daa.com.au/pub/james/python)

В Linux у разработчиков есть интерфейсы Python к соответствующим библиотекам GUI, лежащим в центре оконных систем KDE и Gnome. Пакеты расширений PyKDE и PyQt предоставляют доступ к библиотекам разработчика KDE (для PyKDE необходим PyQt). Пакеты расширений gnome-python и PyGTK экспортируют вызовы инструментария Gnome и GTK для использования в сценариях Python (gnome-python требует PyGTK). Оба эти набора расширений переносимы в той же мере, что и используемые ими библиотеки. На сегодняшний день библиотека классов Qt работает в Unix и Windows, KDE работает на платформах Linux и Unix, а GTK и Gnome работают на платформах Linux и Unix (хотя разрабатывается перенос GTK на Windows). Новейшие подробности можно найти на соответствующих сайтах.

MFC (http://www.python.org/windows)

Пакет расширений Windows *win32all.exe* для Python, который можно взять на сайте Python и с компакт-диска, прилагаемого к данной книге, содержит оболочки для структуры Microsoft Foundation Classes (MFC) – библиотеки разработчика, содержащей компоненты интерфейса пользователя. С помощью расширений Windows программы Python могут строить GUI в Windows с помощью тех же вызовов MFC, которые используются в языках типа Visual C++. В пакет расширений входит Pythonwin – пример программы, использующей MFC, реализующий GUI для разработки на Python. Это специфическое для Windows решение, но оно может оказаться привлекательным для разработчиков, сделавших ранее ставку на использование структуры MFC из Visual C++.

WPY (http://www.python.org/ftp/python/wpy)

Подобная MFC библиотека GUI для Python, перенесенная для выполнения и в X Windows для Unix (где она использует Tk), и в Windows для PC (где она использует MFC). Сценарии WPY без изменений работают на каждой из платформ, но используют стиль кодирования MFC.

X11 (http://www.cwi.nl/ftp/sjoerd/index.html)

Для Python существуют также интерфейсы непосредственно к библиотекам X Windows и Motif. Они предоставляют максимум контроля над средой разработки X11, но являются решением только для X.

Существуют и другие менее известные инструментарии GUI для Python, и к тому времени, когда вы будете читать эту книгу, появятся, вероятно, новые (например, объявленный недавно перенос Python в структуру .NET в Windows, возможно, также предоставит средства для интерфейса пользователя).[1] Кроме того, названия пакетов и

[1] В части III «Сценарии для Интернета» мы научимся строить интерфейсы пользователя внутри веб-броузера. Пока мы сосредоточимся на более традиционных GUI, которые могут и не быть соединенными с сетью.

сайтов, фигурирующие в данном списке, изменяются в течение времени. Свежий список имеющихся инструментов можно найти на *http://www.python.org* и новом сайте пакетов сторонних разработчиков «Vaults of Parnassus» на *http://www.vex.net/parnassus*.

Обзор Tkinter

Однако все эти варианты GUI сегодня значительно опережает Tkinter как стандарт де-факто реализации пользовательских интерфейсов на Python, которому и посвящена эта часть книги. Переносимость Tkinter, простота получения и доступность, документированность и расширения делают его наиболее широко используемым решением для Python в области GUI в течение многих лет. Вот несколько причин этого:

Доступность

Tkinter в целом рассматривается как облегченный набор инструментов и одно из наиболее простых решений GUI для Python из имеющихся на сегодняшний день. В отличие от более крупных структур, можно легко сразу начать работать с Tkinter, не осваивая предварительно значительно более крупных моделей взаимодействия классов. Как будет показано, программист может создать простой интерфейс GUI Tkinter в нескольких строках кода Python и постепенно наращивать мощь до написания GUI промышленного качества.

Переносимость

Сценарий Python, в котором GUI строится с помощью Tkinter, будет работать без изменений в исходном коде на всех основных современных оконных платформах: Microsoft Windows, X Windows (в Unix и Linux) и Macintosh. Более того, этот сценарий создаст интерфейс, внешний вид которого будет привычен пользователям каждой из этих платформ. Сценарий Python/Tkinter выглядит, как программа для Windows в Windows; в Unix и Linux он обеспечивает такое же взаимодействие с пользователем, но демонстрирует вид, знакомый пользователям X Windows; и на Mac он выглядит так, как должна выглядеть программа Mac.

Простота получения

Tkinter является стандартным модулем библиотеки Python, поставляемой вместе с интерпретатором. Если у вас есть Python, то у вас есть и Tkinter. Более того, с большинством пакетов установки Python (в том числе стандартной самоустанавливающейся программой Python для Windows) увязана поддержка Tkinter. Благодаря этому сценарии, написанные с использованием модуля Tkinter, сразу могут работать с большинством интерпретаторов Python, не требуя дополнительных действий по установке.[1] Tkinter также в целом лучше поддерживается, чем существующие сегодня альтернативные для него пакеты. Поскольку используемая в нем библиотека Tk используется также языками программирования Tcl и Perl, ей уделяется больше внимания и усилий разработчиков, чем другим имеющимся инструментариям.

---

[1] Некоторые дистрибутивы Python для Unix-подобных платформ все еще не увязаны с поддержкой Tk, поэтому может потребоваться установка его на вашу машину. На некоторых платформах Unix и Linux может также потребоваться настройка переменных оболочки для библиотеки Tcl/Tk, чтобы иметь возможность использовать Tkinter. Подробности относительно установки смотрите в приложении B; обычно можно обойтись без подробностей компиляции Tk, если найти другой дистрибутив Python с привязкой к Tk (например, RPM для Linux).

Естественно, при использовании инструментария GUI играют роль и другие факторы, такие как документация и расширения. Рассмотрим вкратце, что можно в этом отношении сказать о Tkinter.

Документация Tkinter

В данной книге исследуются основы Tkinter и большинство средств графических элементов, чего должно хватить для начала разработки реальных GUI на Python. С другой стороны, книга не является полным справочником по библиотеке Tkinter. К счастью, когда я пишу этот параграф, в продаже есть по крайней мере одна книга, посвященная использованию Tkinter в Python, и готовятся другие (смотрите список книг по Python на *http://www.python.org*). Кроме книг можно найти электронную документацию по Tkinter; полный комплект руководств по Tkinter в настоящее время есть на сайте *http://www.pythonware.com/library*.

Кроме того, поскольку инструментарий Tk, используемый в Tkinter, также является стандартом де-факто в сообществе open source для языков сценариев в целом, можно использовать и другие источники документации. Например, поскольку библиотека Tk принята также к использованию в языках программирования Tcl и Perl, книги и документация по Tk, написанные для этих двух языков, могут также непосредственно использоваться для Python/Tkinter (хотя и с определенным учетом синтаксиса).

Честно говоря, я изучал Tkinter по книгам и справочникам Tcl/Tk – просто замените строки на Tcl объектами Python, и вы получите в свое распоряжение дополнительные библиотеки справочников (чтение документации по Tk облегчит руководство по преобразованию Tk в Tkinter, помещенное в табл. 6.2 в конце данной главы). Например, «Tcl/Tk Pocket Reference» («Карманный справочник по Tcl/Tk») O'Reilly может служить прекрасным дополнением к учебному материалу по Tkinter в данной части книги. Кроме того, поскольку понятия Tk знакомы большому числу программистов, поддержку по Tk можно легко получить в Сети.

Расширения Tkinter

Благодаря широкому использованию Tkinter программистам доступны готовые расширения Python, предназначенные для работы с ним или дополняющие его. Например:

PIL (http://www.pythonware.com/products/pil/)

Python Imaging Library является пакетом расширения с открытым исходным кодом, содержащим дополнительные средства для работы с графикой в Python. Помимо всего прочего, он расширяет базовый набор графических объектов Tkinter, добавляя поддержку вывода многих типов графических файлов (подробности смотрите в разделе «Графические изображения» в конце главы 7). Кроме разработки PIL, PythonWare создает также средства разработки GUI для программирования на Python и Tkinter, имеющие название PythonWorks; за подробностями обратитесь на сайт компании.

PMW (http://www.dscpl.com.au/pmw)

Python Mega Widgets является инструментальным набором расширений для создания в Python составных графических элементов высокого уровня с помощью модуля Tkinter. Он расширяет API Tkinter с помощью набора более сложных графических элементов для разработки продвинутых GUI и системы, позволяющей создавать собственные элементы. В число готовых и расширяемых графических элементов, имеющихся в пакете, входят блокноты, комбинированные списки, элементы для выбора, диалоговые окна, окна кнопок и интерфейс к пакету Blt для

построения графов. Интерфейс к графическим элементам («megawidgets») PMW аналогичен интерфейсу базовых графических элементов Tkinter, поэтому в сценариях Python могут совместно использоваться элементы PMW и стандартные элементы Tkinter.

IDLE (входит в поставку Python)

Интегрированная среда разработки Python IDLE сама написана на Python и Tkinter и поставляется и устанавливается вместе с пакетом Python (если у вас свежий интерпретатор Python, то должна быть и среда IDLE – в Windows щелкните по кнопке Пуск, выберите меню Программы, щелкните по элементу Python и вы увидите ее). Как описано в приложении B, в IDLE есть редакторы кода Python с выделением синтаксиса цветом, отладка с интерфейсом «point-and-click» («укажи и щелкни») и многое другое. Эта среда служит примером использования Tkinter.

Если вы собираетесь заниматься коммерческой разработкой GUI с помощью Tkinter, вам, вероятно, нужно ознакомиться с такими расширениями, как PMW и PIL, после изучения основ Tkinter в данной книге. Они могут сократить время разработки и добавить блеска в ваши GUI. Самые последние детали и ссылки смотрите на посвященных Python сайтах, указанных выше.

Структура Tkinter

С более технической точки зрения, Tkinter является интегрирующей системой, требующей некоторой особой структуры программы. Что это означает для кода, мы увидим чуть позднее, а пока кратко познакомимся с некоторыми терминами и понятиями, лежащими в основе программирования GUI в Python.

Строго говоря, Tkinter является просто названием интерфейса Python к Tk – библиотеке GUI, первоначально написанной для использования с языком программирования Tcl и разработанной создателем Tcl Джоном Аустерхаутом (John Ousterhout). Модуль Python Tkinter связывается с Tk, а API Tk, в свою очередь, имеет интерфейс с используемой оконной системой: Microsoft Windows, X Windows в Unix или Macintosh.

Tkinter для Python добавляет поверх Tk программный слой, позволяющий сценариям Python обращаться к Tk и строить и настраивать интерфейсы, возвращая управление обратно в сценарии Python, которые обрабатывают события, генерируемые пользователем (например, щелчки мышью). Таким образом, *вызовы* GUI внутренне направляются из сценария Python в Tkinter, а затем в Tk; *события* GUI направляются из Tk в Tkinter, а затем обратно в сценарий Python. В части V «Интеграция» мы встретимся с этими пересылками под теми именами, которые они имеют в интеграции с C: расширение (*extending*) и встраивание (*embedding*).[1]

К счастью, программистам Python обычно не нужно беспокоиться обо всей этой происходящей внутри маршрутизации вызовов: они просто создают графические элементы и регистрируют функции Python для обработки событий этих элементов. Однако эта общая структура служит причиной того, что обработчики событий обычно называют обработчиками обратных вызовов (*callback handlers*), поскольку библиотека GUI «делает обратный вызов» кода Python, когда происходят события.

---

[1] Раз уж я коснулся этого: структура Tkinter состоит из написанного на Python файла модуля `Tkinter` и модуля расширения под названием `_tkinter`, написанного на C. `_tkinter` имеет интерфейс к библиотеке Tk и направляет обратные вызовы назад объектам Python с помощью средств встраивания; `Tkinter` добавляет поверх `_tkinter` интерфейс, основанный на классах. Однако в сценариях нужно всегда импортировать `Tkinter` (а не `_tkinter`); последний является модулем реализации только для внутреннего использования (поэтому он имеет необычное имя).

В действительности мы обнаружим, что программы Python/Tkinter полностью *управляются событиями*: они строят экранные представления и регистрируют обработчики событий, а затем ничего не делают и только ждут, когда произойдут события. Во время этого ожидания библиотека GUI Tk выполняет цикл событий (*event loop*), который следит за щелчками мыши, нажатием клавиш и т. п. Вся обработка, выполняемая прикладной программой, происходит в зарегистрированных обработчиках обратных вызовов в ответ на происходящие события. Кроме того, вся информация, необходимая одновременно разным событиям, должна храниться по ссылкам с длительным сроком жизни, например в глобальных переменных и атрибутах экземпляров классов. Представление об обычной линейной логике выполнения программы неприменимо в области GUI: здесь нужно мыслить на языке небольших фрагментов кода.

Библиотека Tk становится в Python *объектно-ориентированной*, поскольку объектно-ориентированным является Python: слой Tkinter экспортирует API Tk как классы Python. Используя Tkinter, можно применять как простой подход на основе вызовов функций для создания графических элементов и интерфейсов, так и объектно-ориентированную технологию, такую как наследование и композиция, для дополнительной настройки и расширения классов из базового набора Tkinter. Большие GUI обычно строятся в Tkinter как деревья связанных между собой графических элементов Tkinter и часто реализуются в виде классов Python, чтобы обеспечивать структурность и сохранять информацию о состоянии в промежутке между событиями. В этой части книги мы увидим, что GUI Tkinter, код которого представляется классами, почти по умолчанию становится многократно используемым программным компонентом.

Взбираясь по кривой обучения GUI-программированию

Перейдем к деталям. Начнем с нескольких небольших примеров, иллюстрирующих базовые понятия, и покажем окна, которые они создают на экране. По ходу изложения примеры будут становиться более сложными.

«Hello World» в четыре строки (или меньше)

Обычно первый пример для систем GUI показывает, как вывести в окне строку «Hello World». Код примера 6.1 делает это в четырех строках.

Пример 6.1. PP2E\Gui\Intro\gui1.py

```
from Tkinter import Label                              # получить объект графического элемента
widget = Label(None, text='Hello GUI world!')          # создать экземпляр
widget.pack()                                          # разместить его
widget.mainloop()                                      # начать цикл событий
```

Это законченная программа Python, использующая GUI Tkinter. При запуске этого сценария получается простое окно с меткой посередине. Его вид в Windows показан на рис. 6.1.

Рис. 6.1. «Hello World» (gui1) в Windows

Пока здесь нечем особенно похвастать, но обратите внимание, что это независимое полнофункциональное окно на экране компьютера. Его можно развернуть на весь экран,

свернуть и спрятать на системной панели, изменить его размер. Щелкните по квадратику «X» в правом верхнем углу окна, чтобы закрыть его и завершить программу.

Кроме того, создающий это окно сценарий полностью переносим – при запуске этого же файла в Linux создается аналогичное окно, но ведет оно себя в соответствии с работающим в Linux менеджером окон. Например, на рис. 6.2 показано, как этот простой сценарий действует в системе Linux X Windows под оконными менеджерами KDE и Gnome, соответственно. Даже в одной и той же операционной системе один и тот же код Python создает разный внешний вид объектов для разных систем окон.

Рис. 6.2. «Hello World» в Linux с KDE и Gnome

Тот же файл сценария будет выглядеть иначе при запуске на Macintosh или в других менеджерах Unix-подобных систем. Однако на всех платформах основные выполняемые функции будут одинаковыми.

Основы кодирования на Tkinter

Сценарий `gui1` является тривиальным примером, но он иллюстрирует шаги, которые выполняются большинством программ Tkinter. Этот код Python делает следующее:

1. Загружает класс графических элементов из модуля `Tkinter`
2. Создает экземпляр импортированного класса `Label`
3. Упаковывает (размещает) новый `Label` в его родительском элементе
4. Вызывает `mainloop`, чтобы показать окно и начать цикл событий Tkinter

Метод `mainloop`, вызываемый последним, помещает метку на экран и входит в *состояние ожидания* Tkinter, в котором отслеживаются события GUI, генерируемые пользователем. Внутри функции `mainloop` Tkinter следит за такими вещами, как клавиатура или мышь, чтобы обнаружить порожденные пользователем события. В такой модели вызов `mainloop` никогда не возвратится в сценарий, пока GUI выводится на экране.[1] Как мы увидим, добравшись до больших сценариев, единственным способом делать что-либо после вызова `mainloop` является регистрация обработчиков обратного вызова для реакции на события.

Обратите внимание, что для открытия GUI этого сценария действительно нужны *оба* шага: 3 и 4. Для того чтобы вообще отобразить окно GUI, нужно вызвать `mainloop`; для вывода графических элементов внутри окна они должны быть упакованы (или организованы иным образом), чтобы менеджер геометрии Tkinter знал о них. На самом деле, если вызвать только `mainloop` или только `pack`, не вызывая второго, окно будет показывать не то, что нужно: `mainloop` без `pack` показывает пустое окно, а `pack` без `mainloop` не показывает ничего, потому что сценарий не войдет в состояние ожидания событий (можете попробовать). Поскольку понятия, иллюстрируемые этим простым примером, лежат в центре большинства программ Tkinter, рассмотрим их несколько глубже, прежде чем двинуться дальше.

---

[1] Формально вызов `mainloop` возвращается в ваш сценарий только после выхода из цикла событий Tkinter. Обычно это происходит при закрытии главного окна GUI, но может случиться и в ответ на явный вызов метода `quit`, который завершает вложенный цикл событий, но в целом оставляет GUI открытым. Почему это имеет значение, вы узнаете в главе 7.

Создание графических элементов

При создании графических элементов в Tkinter можно указать, как они должны быть настроены. Сценарий `gui1` передает два аргумента конструктору класса `Label`:

- Первый аргумент задает объект *родительского графического элемента*, к которому нужно прикрепить новую метку. В данном случае `None` означает «прикрепить новый `Label` к установленному по умолчанию окну верхнего уровня данной программы». Позднее в этой позиции мы будем задавать реальные графические элементы, чтобы прикреплять метки к другим объектам-контейнерам.
- Второй аргумент служит параметром конфигурации для `Label`, передаваемым в формате ключевого слова: параметр `text` задает строку текста, которая должна появиться в виде сообщения на метке. Большинство конструкторов графических элементов принимает несколько аргументов с ключевыми словами для задания ряда параметров (цвет, размер, обработчики обратного вызова и т. д.). Однако большинство параметров конфигурации графических элементов имеет разумные значения по умолчанию для каждой платформы, чем в значительной мере объясняется простота Tkinter – большинство параметров нужно устанавливать только тогда, когда желательно сделать пользовательские настройки.

Как будет показано, аргумент, задающий родительский графический элемент, является основой, используемой для построения сложных GUI в виде деревьев графических элементов. Tkinter действует по принципу «что построишь, то и получишь»: деревья объектов графических элементов создаются как модели того, что мы хотим видеть на экране, а затем мы просим дерево изобразить себя с помощью вызова `mainloop`.

Менеджеры геометрии

Метод `pack` графического элемента, к которому обращается сценарий `gui1`, вызывает *менеджер геометрии – упаковщик* (*packer*) – один из трех способов управления организацией графических элементов в окне. Менеджеры геометрии Tkinter просто организуют один или несколько графических элементов в контейнере (иногда называемом родителем или хозяином). Как окна, так и фреймы (кадры) верхнего уровня (особый вид графического элемента, с которым мы познакомимся позднее) могут служить контейнерами, а контейнеры могут вкладываться внутрь других контейнеров, образуя иерархические отображения.

Упаковывающий менеджер геометрии использует значения опции *constraint* (ограничения) для автоматического размещения графических элементов в окне. Сценарии дают команды высокого уровня (например, «прикрепить данный графический элемент к верхней части контейнера и растянуть по вертикали до заполнения пространства»), а не абсолютные координаты в пикселах. Поскольку такие ограничения весьма абстрактны, упаковщик предоставляет мощную и простую в использовании систему расположения элементов. В действительности вам даже не нужно задавать ограничения – если не передавать аргументы в `pack`, выполняется упаковка по умолчанию, при которой графический элемент прикрепляется к верхнему краю.

Мы неоднократно будем встречаться с упаковщиком в этой главе и использовать во многих примерах книги. В главе 8 мы также познакомимся с альтернативным менеджером геометрии `grid` и системой размещения графических элементов в контейнере в табличном виде (то есть по строкам и колонкам). Третья альтернатива, менеджер геометрии *placer*, описывается в документации Tk и не описывается в данной книге: он менее популярен, чем менеджеры `pack` и `grid`, и в крупных GUI его использование может быть затруднительно.

Выполнение программ GUI

Как и любой код Python, модуль примера 6.1 может быть запущен несколькими способами: путем выполнения в качестве программного файла верхнего уровня:

```
C:\...\PP2E\Gui\Intro>python gui1.py
```

путем импорта из сеанса Python или другого файла модуля:

```
>>> import gui1
```

путем выполнения как исполняемого файла Unix, если добавить в начале особую строку, начинающуюся с `#!`:

```
% gui1.py &
```

и любым другим способом, которым программы Python могут быть запущены на вашей платформе. Например, этот сценарий может быть также выполнен путем щелчка по имени файла в проводнике Windows, или его код может быть введен интерактивно в ответ на приглашение `>>>`. Можно даже выполнить его из программы на C, вызвав соответствующую функцию API для встраивания (подробности можно найти в главе 20 «Встраиваем Python»).

Иными словами, практически нет специальных правил, согласно которым должен запускаться код GUI на Python. Интерфейс Tkinter (и сам Tk) соединен с интерпретатором Python. Когда программа Python вызывает функции GUI, они просто за кулисами передаются встроенной системе GUI. Это облегчает написание инструментов командной строки, которые вызывают появление всплывающих окон; они выполняются так же, как чисто текстовые сценарии, которые изучались в предыдущей части книги.

Как избежать появления консолей DOS в Windows

Ранее мы узнали, что если имя программы имеет расширение *.pyw*, а не *.py*, перенос Python в Windows не показывает окно консоли DOS, которое служит в качестве стандартных потоков файла, запускаемого щелчком по значку. Теперь, когда мы наконец-то начали создавать собственные окна, этот прием с именем файла становится еще более полезным.

Для того чтобы видеть только окна, создаваемые сценарием независимо от способа его запуска, дайте файлам своих сценариев GUI расширение *.pyw*, если они будут выполняться под Windows. Например, щелчок по файлу примера 6.2 в проводнике Windows создает *только* окно, показанное на рис. 6.1.

Пример 6.2. PP2E\Gui\Intro\gui1.pyw

```
...то же, что gui1.py...
```

Можно также избежать показа всплывающих окон DOS в Windows, запуская программу с исполняемым файлом *pythonw.exe* вместо *python.exe* (в действительности файлы *.pyw* просто зарегистрированы для открытия посредством *pythonw*). В Linux расширение *.pyw* не мешает, но и не является необходимым – на Unix-подобных машинах нет понятия всплывающих окон для отображения потоков. С другой стороны, если в будущем потребуется выполнять ваши сценарии GUI под Windows, добавление «w» в конце имени может освободить от необходимости каких-то дополнительных действий по переносу. В данной книге имена файлов *.py* иногда используются для вызова окон консоли, чтобы видеть сообщения, выводимые в Windows.

Варианты кодирования в Tkinter

Как можно предполагать, есть разные способы написания кода примера gui1. Например, если вы хотите более явным образом описать в своем сценарии импорт из Tkinter, возьмите весь модуль и добавляйте префикс из имени модуля ко всем относящимся к нему именам, как в примере 6.3.

Пример 6.3. PP2E\Gui\Intro\gui1b.py – import против from

```
import Tkinter
widget = Tkinter.Label(None, text='Hello GUI world!')
widget.pack()
widget.mainloop()
```

В реальных примерах это может быть утомительным – Tkinter экспортирует десятки классов графических переменных и констант, которые повсюду встречаются в сценариях GUI Python. На самом деле обычно проще использовать *, чтобы одним махом импортировать все, находящееся в модуле Tkinter. Это показано в примере 6.4.

Пример 6.4. PP2E\Gui\Intro\gui1c.py – корень, край, упаковка вместе

```
from Tkinter import *
root = Tk()
Label(root, text='Hello GUI world!').pack(side=TOP)
root.mainloop()
```

Модуль Tkinter старается экспортировать только то, что действительно требуется, поэтому он один из немногих, для которых формат импорта * можно применять относительно безопасно.[1] К примеру, константа TOP в вызове pack является одним из многих имен, экспортируемых модулем Tkinter. Это просто имя переменной (TOP="top"), которой заранее присвоено значение в Tkconstants – модуле, автоматически загружаемом Tkinter.

При упаковке графических элементов можно указать, к которому краю родительского контейнера они должны быть прикреплены – TOP, BOTTOM, LEFT или RIGHT. Если параметр side в pack не посылается (как в предшествующих примерах), графический элемент по умолчанию прикрепляется к верхнему краю (TOP). В целом, более крупные GUI могут строиться в Tkinter как набор прямоугольников, прикрепляемых к надлежащим сторонам других, охватывающих их прямоугольников. Как будет показано позднее, Tkinter располагает графические элементы в прямоугольнике в соответствии с очередностью их упаковки и параметром прикрепления side. Если графические элементы располагаются по сетке, им присваиваются номера строк и колонок. Однако все это имеет смысл, только если в окне больше одного графического элемента, поэтому продолжим наше изложение.

Обратите внимание, что в этой версии метод pack вызывается сразу после создания метки, которая не присваивается какой-либо переменной. Если нет необходимости сохранять графический элемент, можно упаковывать его таким способом на месте, чем сэкономить строку кода. Такая форма будет использоваться, если графический элемент прикрепляется к более крупной структуре и затем никогда больше не используется. Могут возникнуть сложности, если присваивать результат, возвращаемый pack, объяснение чего откладывается до того времени, когда будут изучены еще некоторые основные понятия.

---

[1] Если рассмотреть файл *Tkinter.py* в библиотеке исходного кода Python, то можно заметить, что имена модулей, не предназначенных для экспорта, начинаются с одного символа подчеркивания. Python не копирует такие имена при обращении к модулю в операторе from в формате *.

Мы также используем здесь в качестве родителя экземпляр класса графических элементов Tk вместо None. Tk представляет главное («корневое») окно программы – то, которое открывается вместе с запуском программы. Tk используется также как родительский графический элемент по умолчанию, когда в другие вызовы графических элементов не передается никакого родителя или в качестве родителя задается None. Иными словами, по умолчанию графические элементы просто прикрепляются к главному окну программы. В данном сценарии это поведение по умолчанию задается явно путем создания и передачи самого объекта Tk. В главе 7 будет показано, как для создания новых всплывающих окон, действующих независимо от главного окна программы, используются графические элементы Toplevel.

В Tkinter некоторые методы графических элементов экспортируются также в виде функций, что позволяет урезать пример 6.5 всего лишь до трех строчек кода.

Пример 6.5. PP2E\Gui\Intro\gui1d.py – минимальный вариант

```
from Tkinter import *
Label(text='Hello GUI world!').pack()
mainloop()
```

Можно вызывать mainloop модуля Tkinter с указанием графического элемента или без него (то есть как функцию или метод). В этом варианте мы не передавали в Label и аргумент родителя: если он опушен, то принимает значение по умолчанию None (что, в свою очередь, по умолчанию означает Tk). Но использование этого значения по умолчанию становится менее удобным при создании более крупных интерфейсов: такие элементы, как метки, чаще прикрепляются к другим контейнерам графических элементов.

Основы изменения размеров графических элементов

Размер окон верхнего уровня – таких, как создававшееся во всех показанных до сих пор вариантах кода, – обычно может изменяться пользователем: нужно просто растянуть окно с помощью мыши. На рис. 6.3 показано, как выглядит увеличенное в размере окно.

Рис. 6.3. Увеличение размера окна gui1

Это не очень хорошо: метка сохраняет положение у верхнего края родительского окна вместо того, чтобы оказаться в середине, но положение легко исправить с помощью пары опций pack, показанных в примере 6.6.

Пример 6.6. PP2E\Gui\Intro\gui1e.py – расширение

```
from Tkinter import *
Label(text='Hello GUI world!').pack(expand=YES, fill=BOTH)
mainloop()
```

При упаковке графических элементов можно указывать, должен ли графический элемент увеличиться в размере (expand), чтобы заполнить все свободное пространство, и если да, то как он должен быть растянут, чтобы заполнить все это пространство. По умолчанию графический элемент не увеличивается, когда это происходит с его роди-

телем. Но в данном сценарии имена YES и BOTH (импортированные из модуля Tkinter) указывают, что эта метка должна увеличиваться вместе со своим родителем, то есть главным окном. Так оно и происходит, как показывает рис. 6.4.

Рис. 6.4. gui1e с изменением размера графического элемента

Технически упаковывающий менеджер геометрии назначает размер каждому отображаемому графическому элементу исходя из его содержимого (длины текстовой строки и т. д.). По умолчанию графический элемент может занимать только отведенное ему пространство и не может быть больше назначенного ему размера. Опции expand и fill позволяют более точно определять тип расширения:

- Опция expand=YES просит упаковщика расширить в целом пространство, выделяемое графическому элементу, чтобы оно заняло все свободное место в родительском контейнере.
- Опция fill используется для растяжения графического элемента, чтобы он занял все выделенное ему пространство.

Сочетания этих двух опций производят различные эффекты расположения и изменения размеров, при этом некоторые из них имеют смысл только при наличии в окне нескольких графических элементов. Например, задание expand без fill размещает графический элемент в центре расширенного пространства, а опция fill может задавать растяжку только по вертикали (fill=Y), только по горизонтали (fill=X) или обе одновременно (fill=BOTH). Задавая для всех графических элементов в GUI эти ограничения и стороны, к которым они прикреплены, можно довольно точно управлять их взаимным расположением. В последующих главах будет показано, что менеджер геометрии grid использует для изменения размеров совершенно другой протокол.

Впервые столкнувшись с этим, можно запутаться, и мы вернемся к этому позднее. Но если вы не уверены в том, какой результат будет иметь некоторая комбинация expand и fill, просто попробуйте, что получится – в конце концов, это Python. А пока запомните, что комбинация expand=YES и fill=BOTH встречается, вероятно, чаще всего и означает «расширить отведенное мне место, чтобы оно занимало все свободное пространство, и растянуть меня так, чтобы заполнить расширенное пространство во всех направлениях». Для нашего примера «Hello World» итоговым результатом будет рост метки при увеличении размеров окна, поэтому метка всегда остается в центре.

Настройка параметров графического элемента и заголовка окна

До сих пор мы сообщали Tkinter, что должно выводиться в метке, передавая ее текст в качестве аргумента ключевого слова при вызове *конструктора* метки. Оказывается, существует еще два способа задания параметров конфигурации метки. В примере 6.7 параметр метки text задается после ее создания путем присвоения его по ключу text графического элемента – графические элементы перегружают операции индексирования, чтобы доступ к параметром можно было осуществлять по ключу, подобно словарям.

Пример 6.7. PP2E\Gui\Intro\guif.py – ключи параметров

```
from Tkinter import *
widget = Label()
widget['text'] = 'Hello GUI world!'
widget.pack(side=TOP)
mainloop()
```

Но чаще параметры графических элементов устанавливаются после их создания путем вызова метода `config`, как в примере 6.8.

Пример 6.8. PP2E\Gui\Intro\gui1g.py – config и заголовки

```
from Tkinter import *
root = Tk()
widget = Label(root)
widget.config(text='Hello GUI world!')
widget.pack(side=TOP, expand=YES, fill=BOTH)
root.title('gui1g.py')
root.mainloop()
```

Метод `config` (для которого можно также использовать его синоним `configure`) можно вызвать в любой момент после создания графического элемента, чтобы на лету изменить его внешний вид. Можно, например, вызвать метод `config` этой метки дальше в сценарии, чтобы изменить текст, который в ней выводится; примеры такой динамической реконфигурации можно найти в последующих примерах в этой части книги.

Обратите также внимание на вызов в этой версии примера метода `root.title` – этот метод устанавливает заголовок, появляющийся вверху окна, как показано на рис. 6.5. Вообще говоря, окна верхнего уровня типа `Tk` `root` в этом примере экспортируют интерфейсы менеджера окон: то, что имеет отношение к границе окна, а не к его содержимому.

Рис. 6.5. gui1g с расширением и заголовком окна

Исключительно для развлечения в этой версии метка также центрируется при изменении размеров окна путем установки параметров `expand` и `fill`. На самом деле здесь почти все делается явно, что лучше соответствует обычному способу кодирования метки в полноценных интерфейсах – с указанием родителя, политики расширения и способа прикрепления, без использования значения по умолчанию.

Еще одна версия в память о былых временах

Наконец, если вы склонны к минимализму и испытываете ностальгию по старому коду на Python, можно запрограммировать «Hello World», как в примере 6.9.

Пример 6.9. PP2E\Gui\Intro\gui1-old.py – вызов словаря

```
from Tkinter import *
Label(None, {'text': 'Hello GUI world!', Pack: {'side': 'top'}}).mainloop()
```

Здесь для создания окна хватило двух строк, хотя они ужасны. Эта схема основана на старом стиле кодирования, широко использовавшемся до Python 1.3, когда параметры конфигурации передавались не как аргументы ключевых слов, а в словаре.[1] В этой схеме параметры упаковщика могут пересылаться как значения по ключу `Pack` (класса в модуле Tkinter).

Схема с использованием в вызове словаря по-прежнему действует, и ее можно обнаружить в старом коде Python, но, пожалуйста, не пользуйтесь ею – применяйте в своих сценариях Tkinter для передачи параметров ключевые слова и явный вызов метода `pack`. Единственная причина, по которой я не выкинул этот пример, та, что словари все еще могут быть полезны, когда нужно динамически вычислить и передать набор параметров. С другой стороны, теперь есть встроенная функция `apply`, которая также позволяет передать явный словарь аргументов ключевых слов в третьей позиции для аргументов, поэтому нет веских причин вообще когда-либо использовать формат вызова со словарем из версий, предшествующих Tkinter 1.3.

Упаковка графических элементов без их сохранения

В *gui1c.py* из примера 6.4 я начал упаковывать метки, не присваивая им имен. Это действует и является совершенно законным стилем кодирования; но поскольку при первом взгляде на такой код начинающие программисты могут смутиться, необходимо более подробно объяснить здесь, *почему* такой код работает.

В Tkinter объекты классов Python соответствуют реальным объектам, выводимым на экран: мы заставляем объект Python создать объект на экране и вызываем методы объекта Python, чтобы настроить этот экранный объект. Благодаря такому соответствию срок жизни объекта Python должен в целом соответствовать сроку жизни сопоставленного с ним объекта на экране.

К счастью, сценариям Python обычно не требуется беспокоиться о времени жизни объекта. В действительности они обычно вообще не обязаны поддерживать ссылки на создаваемые объекты графических элементов, если только не собираются в дальнейшем изменять конфигурацию этих объектов. Например, при программировании с использованием Tkinter часто упаковывают графический элемент сразу после его создания, если в дальнейшем ссылка на него не понадобится:

```
Label(text='hi').pack()                         # OK
```

Это выражение вычисляется, как обычно, слева направо: оно создает новую метку и затем немедленно вызывает метод `pack` нового объекта, чтобы установить его расположение при отображении на экране. Обратите, однако, внимание, что в этом выражении объект Python `Label` является временным: так как он не присваивается имени, Python должен убрать его как мусор (уничтожить и освободить память) сразу после выполнения метода `pack`.

Однако поскольку Tkinter осуществляет при создании объектов вызовы Tk, метка будет нарисована на экране как положено, несмотря на то что мы не сохранили в своем сценарии соответствующий объект Python. В действительности Tkinter внутренне связывает объекты графических элементов в дерево, которое живет долго и используется для представления отображаемого на экране, поэтому объект `Label`, созданный

[1] В действительности передача аргументов по ключевым словам впервые была введена в Python, чтобы сделать ясными такие вызовы Tkinter. Внутренне аргументы с ключевыми словами на самом деле передаются как словарь (который можно получить с помощью формата аргумента `**name` в заголовке `def`), поэтому обе схемы аналогичны по реализации. Но между ними большое различие в объеме символов, которые нужно ввести и отладить.

этим оператором, в действительности сохраняется, хотя и не тем кодом, который нами написан.[1]

Иными словами, сценариям обычно не нужно беспокоиться о сроке жизни объектов графических элементов, и вполне допускается создавать графические элементы и сразу упаковывать их в одном и том же операторе. Но это не значит, что можно говорить такие вещи:

```
widget = Label(text='hi').pack()                # неверно!
...использование графического элемента...
```

Кажется, что такая команда должна присвоить только что упакованную метку имени `widget`, но этого не произойдет. На самом деле это известная ошибка новичков в Tkinter. Метод `pack` графического элемента упаковывает его, но не возвращает упакованный графический элемент. В действительности `pack` возвращает объект Python `None`; после такого операции `widget` окажется ссылкой на `None` и все дальнейшие операции с графическим элементом, использующие это имя, окажутся безуспешными. Например, по той же причине окажется неуспешной следующая команда:

```
Label(text='hi').pack().mainloop()              # неверно!
```

Поскольку `pack` возвращает `None`, запрос его атрибута `mainloop` генерирует исключительную ситуацию (и правильно!). Если вы действительно хотите упаковать графический элемент и сохранить ссылку на него, скажите вместо этого:

```
widget = Label(text='hi')                       # тоже OK
widget.pack()
...использование графического элемента...
```

Такой формат несколько многословнее, но не столь замысловат, как упаковка графического элемента в том же операторе, который его создает, и позволяет сохранить графический элемент для последующей обработки. С другой стороны, сценарии, создающие структуру графических элементов, часто добавляют их раз и навсегда, не нуждаясь в последующей их настройке; присвоение долго живущих имен в такой программе является бессмысленным и необязательным.[2]

Добавление кнопок и обратных вызовов

Пока мы научились только выводить сообщения в метках и познакомились попутно с базовыми понятиями Tkinter. Метки хороши для обучения основам, но от пользова-

---

[1] Бывшим программистам на Tcl может быть интересно узнать, что Python не только внутренне строит дерево графических элементов, но и использует его для автоматического создания путей графических элементов, которые вручную кодируются в Tcl/Tk (например, `.panel.row.cmd`). Python использует адреса объектов классов графических элементов для помещения в составляющие пути и записывает названия путей в дереве графических элементов. Например, метке, прикрепленной к контейнеру, может быть присвоено в Tkinter внутреннее имя типа `.8220096.8219408`. Однако вас это не должно волновать – просто создавайте и связывайте объекты графических элементов, передавая их родителей, и предоставьте Python работать с деталями имен путей, основываясь на дереве объектов. Дополнительно об отношениях Tk и Tkinter читайте в конце главы.

[2] В главе 7 мы встретимся с двумя исключениями из этого правила. Сценарии должны самостоятельно сохранять ссылки на объекты графических изображений, потому что соответствующие графические данные теряются при попадании объекта графического изображения Python в уборку мусора. Объекты класса *Variable* Tkinter временно деинициализируют ассоциированную переменную Tk при освобождении занятой памяти, но это встречается нечасто и не так вредно.

тельских интерфейсов обычно требуется несколько большее – способность действительно реагировать на действия пользователя. Программа примера 6.10 создает окно, показанное на рис. 6.6.

Рис. 6.6. Кнопка вверху

Пример 6.10. PP2E\Gui\Intro\gui2.py

```
import sys
from Tkinter import *
widget = Button(None, text='Hello widget world', command=sys.exit)
widget.pack()
widget.mainloop()
```

Здесь мы создаем вместо метки экземпляр класса Tkinter `Button`. Как и прежде, он прикрепляется по умолчанию к верхнему краю `TOP` окна верхнего уровня. Но главное, на что нужно обратить здесь внимание, это аргументы конфигурации кнопки: для параметра с именем `command` значением устанавливается функция `sys.exit`.

В кнопках параметр `command` является тем местом, где задается функция обработчика обратного вызова, выполняемого в дальнейшем при нажатии кнопки. Фактически с помощью `command` *регистрируется* действие, которое должен вызвать Tkinter, когда произойдет событие графического элемента. Обработчик обратного вызова, который здесь использован, не очень интересен: как известно из более ранних глав, встроенная функция `sys.exit` просто прекращает выполнение вызвавшей ее программы. В данном случае это означает, что при нажатии на эту кнопку окно исчезнет.

Так же как и для меток, есть разные способы кодирования кнопок. Пример 6.11 представляет версию, в которой кнопка упаковывается тут же без присвоения ей имени, явным образом прикрепляется к левому краю родительского окна и определяет `root.quit` в качестве обработчика обратного вызова – стандартный метод объекта `Tk`, закрывающий GUI и таким образом завершающий программу (в действительности он завершает вызов `mainloop` цикла событий).

Пример 6.11. PP2E\Gui\Intro\gui2b.py

```
from Tkinter import *
root = Tk()
Button(root, text='press', command=root.quit).pack(side=LEFT)
root.mainloop()
```

Эта версия создает окно, показанное на рис. 6.7. Мы не потребовали от кнопки, чтобы она расширялась на все свободное пространство – она этого и не делает.

Рис. 6.7. Кнопка слева

В обоих последних примерах нажатие на кнопку завершает выполнение программы GUI. В более старом коде Tkinter иногда можно увидеть, как параметру `command` присваивается строка «exit», что вызывает закрытие GUI при нажатии кнопки. При этом используется определенное средство, имеющееся в библиотеке Tk, что менее близко Python, чем `sys.exit` или `root.quit`.

Еще раз об изменении размеров графических элементов: расширение

Даже для такого простого GUI есть много способов определить его внешний вид с помощью основанного на ограничениях менеджера геометрии Tkinter `pack`. Например, чтобы поместить кнопку в центре окна, добавьте в вызов метода `pack` параметр `expand=YES`, и получится окно, как на рис. 6.8. При этом упаковщик выделяет кнопке все свободное пространство, но не растягивает ее до размеров этого пространства.

Рис. 6.8. side=LEFT, expand=YES

Если вы хотите, чтобы кнопке было отдано все свободное пространство и она была растянута по горизонтали, добавьте в вызов `pack` аргументы ключевых слов `expand=YES` и `fill=X`, и получится то, что изображено на рис. 6.9.

Рис. 6.9. side=LEFT, expand=YES, fill=X

В результате кнопка первоначально займет все окно (выделенное ей место расширено, а сама она растянута, чтобы заполнить выделенное пространство). При этом кнопка будет расширяться, если увеличивать размеры родительского окна. Как показано на рис. 6.10, кнопка в этом окне будет расширяться при расширении родителя, но только по горизонтальной оси X.

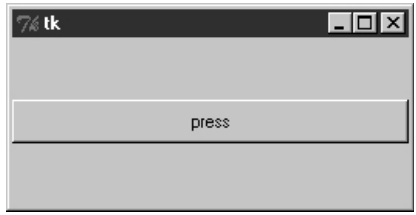


Рис. 6.10. Изменение размера при expand=YES, fill=X

Для того чтобы кнопка расширялась в обоих направлениях, укажите в вызове `pack` параметры `expand=YES` и `fill=BOTH`; теперь при изменении размеров окна кнопка расширяется во все стороны, как показано на рис. 6.11. В действительности, если раскрыть окно на весь экран, получится одна очень большая кнопка Tkinter.

Рис. 6.11. Изменение размеров окна при expand=YES, fill=BOTH

В более сложных интерфейсах такая кнопка будет расширяться только тогда, когда задано расширение для всех содержащих ее графических элементов. В данном случае единственным родителем кнопки служит корневое окно Tk программы, поэтому вопрос о расширяемости родителя пока не стоит. Мы еще вернемся к упаковывающему менеджеру геометрии, когда далее встретимся с интерфейсами, содержащими несколько графических элементов, и еще раз – когда будем изучать альтернативный вызов grid в главе 8.

Добавление определяемых пользователем обработчиков обратного вызова

В простых примерах с кнопкой из предыдущего раздела обработчиком обратного вызова служила уже существующая функция, убивающая программу GUI. Не стоит большого труда зарегистрировать обработчик обратного вызова, который выполняет что-нибудь более полезное. В примере 6.12 определим самостоятельный обработчик обратного вызова в Python.

Пример 6.12. PP2E\Gui\Intro\gui3.py

```
from Tkinter import *

def quit():                                  # пользовательский обработчик обратного вызова
    print 'Hello, I must be going...'        # закрыть окна и процессы
    import sys; sys.exit()

widget = Button(None, text='Hello event world', command=quit)
widget.pack()
widget.mainloop()
```

Создаваемое этим сценарием окно показано на рис. 6.12. Этот сценарий и его GUI почти совпадают с предыдущим примером, но здесь параметр command задает *функцию*, которая определена локально. При нажатии кнопки Tkinter вызывает для обработки события функцию quit в этом файле. В quit оператор print выводит сообщение в поток stdout программы, и процесс GUI завершается, как ранее.

Рис. 6.12. Кнопка, выполняющая функцию Python

Как обычно, stdout является окном, из которого была запущена программа, если только поток не переадресован в файл. Это будет всплывающая консоль DOS, если запустить программу щелчком по ней в Windows – добавьте вызов raw_input перед sys.exit, если всплывающее окно исчезает прежде, чем удается посмотреть сообщение. Вот как выглядит вывод в стандартном потоке при нажатии на кнопку; он генерируется функцией Python, которую автоматически вызывает Tkinter:

```
C:\...\PP2E\Gui\Intro>python gui3.py
Hello, I must be going...

C:\...\PP2E\Gui\Intro>
```

Обычно такие сообщения должны выводиться в другом окне, но мы еще не добрались до того, как это сделать. Конечно, функции обратного вызова обычно выполняют больше действий (и могут даже вызывать вывод новых окон), но основы их использования продемонстрированы в данном примере.

В целом обработчики обратного вызова могут быть любым вызываемым объектом – функциями, анонимными функциями, генерируемыми с помощью лямбда-выражений, связанными методами экземпляров классов или типов или экземплярами классов, наследующими метод перегрузки оператора `__call__`. При обратных вызовах `Button` обработчики не получают никаких аргументов (кроме `self` в связанных методах).

Лямбда-обработчики обратных вызовов

Чтобы несколько конкретизировать последний параграф, вкратце рассмотрим другие методы кодирования обработчика обратного вызова в этом примере. Напомним, что лямбда-выражения в Python при выполнении генерируют новые безымянные объекты функций. Если нужно, чтобы в функцию обработчика передавались дополнительные данные, можно регистрировать лямбда-выражения со значениями аргументов по умолчанию, задающими требуемые дополнительные данные. Как этим воспользоваться, будет показано далее в этой части книги, но для иллюстрации основных идей в примере 6.13 показывается, что получится, если использовать в коде лямбду, а не `def`.

Пример 6.13. PP2E\Gui\Intro\gui3b.py

```
from Tkinter import *
from sys import stdout, exit                # лямбда генерирует функцию,
widget = Button(None,                       # но содержит просто выражение
             text='Hello event world',
             command=(lambda: stdout.write('Hello lambda world\n') or exit()) )
widget.pack()
widget.mainloop()
```

В этом коде есть некоторая хитрость, поскольку лямбды могут содержать только одно выражение; для эмуляции первоначального сценария используется оператор `or`, чтобы заставить выполниться два выражения, и пишет в `stdout`, чтобы имитировать `print`. Обычно же лямбды используются для передачи дополнительных данных в обработчик обратного вызова с помощью значений по умолчанию:

```
def handler(X, Y):              # обычно должен вызываться без аргументов
  здесь используются первоначальные X и Y...

X = здесь что-то...
Y = здесь еще что-то...
Button(text='ni', command=(lambda save1=X, save2=Y: handler(save1, save2)))
```

Хотя Tkinter обращается к обратным вызовам `command` без аргументов, с помощью такой лямбды можно создать косвенную анонимную функцию, которая служит оболочкой для вызова действительного обработчика и передает информацию, существовавшую в момент первоначального создания GUI. Поскольку аргументы по умолчанию вычисляются и сохраняются, когда выполняется лямбда (а не тогда, когда в дальнейшем происходит обращение к ее результату), они дают возможность запомнить объекты, к которым нужно обратиться позднее во время обработки события. Далее мы увидим, как конкретно это может быть использовано. Если синтаксис приводит вас в замешательство, запомните, что лямбда-выражение, типа приведенного в предыдущем коде, обычно может быть закодировано просто в виде вложенных операторов `def`:

```
X = здесь что-то...
Y = здесь еще что-то...
def func(save1=X, save2=Y): handler(save1, save2)
Button(text='ni', command=func)
```

Связанные методы как обработчики обратного вызова

Связанные методы класса тоже очень хорошо работают в качестве обработчиков обратного вызова: они указывают как на экземпляр, которому должно быть послано событие, так и на связанный с ним метод, который должен быть вызван. Предварительно рассмотрим пример 6.14, в котором пример 6.12 переработан так, что регистрирует связанный метод класса вместо функции или результата лямбды.

Пример 6.14. PP2E\Gui\Intro\gui3c.py

```
from Tkinter import *

class HelloClass:
    def __init__(self):
        widget = Button(None, text='Hello event world', command=self.quit)
        widget.pack()
    def quit(self):
        print 'Hello class method world'    # self.quit является связанным методом
        import sys; sys.exit()              # сохраняет пару self+quit

HelloClass()
mainloop()
```

При нажатии кнопки Tkinter вызывает метод этого класса `quit` как обычно, без аргументов. Но в действительности он получает один аргумент – первоначальный объект `self` – хотя Tkinter не передает его явно. Поскольку связанный метод `self.quit` содержит и `self` и `quit`, он совместим с простым вызовом функции; Python автоматически передает аргумент `self` функции метода. Напротив, регистрация несвязанного метода типа `HelloClass.quit` не станет работать, потому что не будет объекта `self`, который можно передать потом при возникновении события.

Позднее мы увидим, что схемы кодирования класса-обработчика обратного вызова тоже предоставляют естественное место для запоминания информации, используемой при возникновении событий: нужно просто присвоить ее атрибутам экземпляра `self`:

```
class someGuiClass:
    def __init__(self):
        self.X = здесь что-то...
        self.Y = здесь что-то еще...
        Button(text='Hi', command=self.handler)
    def handler(self):
        используйте здесь self.X, self.Y...
```

Поскольку событие будет отправлено методу этого класса со ссылкой на первоначальный объект экземпляра, `self` предоставляет доступ к атрибутам, содержащим исходные данные.

Объекты исполняемых классов как обработчики обратного вызова

Поскольку объекты экземпляров классов Python также могут *исполняться*, если они наследуют метод `__call__` для перехвата этой операции, можно также передавать их в качестве обработчиков обратного вызова, как в примере 6.15.

Пример 6.15. PP2E\Gui\Intro\gui3d.py

```
from Tkinter import *

class HelloCallable:
    def __init__(self):                      # запуск __init__ при создании объекта
        self.msg = 'Hello __call__ world'
```

```
    def __call__(self):
        print self.msg                          # выполнение __call__ при вызове в дальнейшем
        import sys; sys.exit()                  # объект класса выглядит как функция

widget = Button(None, text='Hello event world', command=HelloCallable())
widget.pack()
widget.mainloop()
```

Здесь экземпляр `HelloCallable`, зарегистрированный в `command`, тоже может вызываться как обычная функция – Python вызывает его метод `__call__` для обработки операции вызова, выполняемой в Tkinter при нажатии кнопки. Обратите внимание, что здесь для хранения информации, используемой при возникновении событий, используется `self.msg`; `self` является исходным экземпляром класса, когда автоматически вызывается специальный метод `__call__`.

Все четыре версии `gui3` создают одинаковое окно GUI, но при нажатии их кнопок выводят в `stdout` различные сообщения:

```
C:\...\PP2E\Gui\Intro>python gui3.py
Hello, I must be going...

C:\...\PP2E\Gui\Intro>python gui3b.py
Hello lambda world

C:\...\PP2E\Gui\Intro>python gui3c.py
Hello class method world

C:\...\PP2E\Gui\Intro>python gui3d.py
Hello __call__ world
```

Для использования каждой из схем кодирования обратного вызова (функция, лямбда, метод класса, вызываемый класс) могут иметься свои причины, но для описания их на менее теоретическом языке необходимо перейти к более крупным примерам.

Другие протоколы обратного вызова в Tkinter

Запомните также на будущее, что использование параметров `command` для перехвата генерируемых пользователем событий нажатия кнопки является лишь одним из способов регистрации обратного вызова в Tkinter. На самом деле существует ряд способов, с помощью которых сценарии Tkinter могут перехватывать события:

Параметр command кнопки

Как только что было показано, события нажатия кнопки перехватываются путем указания вызываемого объекта в параметре графического элемента `command`. То же относится к другим графическим элементам, похожим на кнопки, с которыми мы познакомимся в главе 7 (например, переключателям и флажкам, ползункам).

Параметры команд меню

В последующих главах обзора Tkinter будет также показано, что параметр `command` используется для задания обработчиков обратного вызова для выбранных пунктов меню.

Протоколы полос прокрутки

Графические элементы полос прокрутки тоже регистрируют обработчики с помощью параметров `command`, но обладают особым протоколом событий, позволяющим им быть взаимно связанными с графическим элементом, для прокрутки которого они предназначены (например, окна списков, вывода текста и холсты): перемещение элемента прокрутки автоматически перемещает графический элемент и наоборот.

Общие методы bind графических элементов

Более общий механизм метода `bind` событий Tkinter может быть использован для регистрации обработчиков обратного вызова событий интерфейсов низкого уровня – нажатий клавиш, перемещений и щелчков мыши и т. д. В отличие от обратных вызовов `command`, обратные вызовы `bind` получают в качестве аргумента объект события (экземпляр класса Tkinter `Event`), который предоставляет контекст события – графический элемент, к которому относится событие, экранные координаты и т. д.

Протоколы менеджера окон

Кроме того, сценарии могут перехватывать события менеджера окон (например, запрос на закрытие окна) путем проникновения в механизм метода `protocol` менеджера окон, который доступен для оконных объектов верхнего уровня: например установка обработчика для `WM_DELETE_WINDOW` перехватывает кнопки, закрывающие окно.

Обратные вызовы планируемых событий

Наконец, сценарии Tkinter могут также регистрировать обработчики обратного вызова, которые должны вызываться в особых контекстах, например при срабатывании таймера, поступлении входных данных и холостом состоянии цикла событий. Сценарии могут также останавливаться и ожидать событий, связанных с изменением состояния окон и специальных переменных. С интерфейсами этих событий мы более подробно познакомимся в конце главы 8.

Связывание событий

Из всех перечисленных способов наиболее общим, но, вероятно, и наиболее сложным является `bind`. Более подробно мы изучим его потом, но для получения первоначального представления приведем пример 6.16, в котором для перехвата нажатий кнопки используется `bind`, а не `command`.

Пример 6.16. PP2E\Gui\Intro\gui3e.py

```
from Tkinter import *

def hello(event):
    print 'Press twice to exit'              # при одинарном щелчке левой кнопкой

def quit(event):                             # при двойном щелчке левой кнопкой
    print 'Hello, I must be going...'        # событие дает графический элемент, x/y и т. д.
    import sys; sys.exit()

widget = Button(None, text='Hello event world')
widget.pack()
widget.bind('<Button-1>', hello)             # связать щелчок левой кнопкой
widget.bind('<Double-1>', quit)              # связать двойной щелчок левой кнопкой
widget.mainloop()
```

Действительно, в этой версии параметр `command` для кнопки вообще не задан. Вместо этого связываются обработчик обратного вызова низкого уровня для щелчков левой кнопкой (`<Button-1>`) и для двойных щелчков левой кнопкой (`<Double-1>`) внутри области отображения кнопки. Метод `bind` принимает большую группу таких идентификаторов событий в разнообразных форматах, с которыми мы познакомимся в главе 7.

При выполнении этого сценария снова создается то же самое окно (см. рис. 6.13). При щелчке по кнопке выводится сообщение, но программа не прекращает выполняться; чтобы, как и раньше, выйти, нужно щелкнуть по кнопке дважды. Вот сообщения, вы-

водимые после двух простых щелчков и одного двойного (двойной щелчок сначала запускает обратный вызов для одиночного щелчка):

```
C:\...\PP2E\Gui\Intro>python gui3e.py
Press twice to exit
Press twice to exit
Press twice to exit
Hello, I must be going...
```

Хотя в этом сценарии щелчки по кнопке перехватываются вручную, конечный результат примерно такой же: специальные протоколы графических элементов типа параметра кнопки `command` в действительности являются просто интерфейсами более высокого уровня к событиям, которые могут также быть перехвачены с помощью `bind`.

Более подробно мы разберем `bind` и все другие способы установки обработчиков обратных вызовов для событий Tkinter далее в этой книге. Однако сначала займемся построением более крупных GUI, состоящих не из одной лишь кнопки, и другими способами использования классов в работе над GUI.

Добавление нескольких графических элементов

Настало время строить интерфейсы пользователя не с одним, а с несколькими графическими элементами. Пример 6.17 создает окно, показанное на рис. 6.13.

Рис. 6.13. Окно с несколькими графическими элементами

Пример 6.17. PP2E\Gui\Intro\gui4.py

```
from Tkinter import *

def greeting():
    print 'Hello stdout world!...'

win = Frame()
win.pack()
Label(win, text='Hello container world').pack(side=TOP)
Button(win, text='Hello', command=greeting).pack(side=LEFT)
Button(win, text='Quit',  command=win.quit).pack(side=RIGHT)

win.mainloop()
```

В этом примере создается графический элемент `Frame` (еще один класс Tkinter), к которому прикрепляются три других графических элемента — `Label` и два `Button` — путем передачи `Frame` в качестве их первого аргумента. На языке Tkinter это означает, что `Frame` становится *родителем* для трех других графических элементов. Обе кнопки этого интерфейса запускают обратный вызов:

- Нажатие на кнопку Hello запускает функцию `greeting`, определенную внутри этого файла, которая осуществляет вывод снова на `stdout`.
- Нажатие на кнопку Quit вызывает стандартный метод Tkinter `quit`, который `win` наследует от класса `Frame` (`Frame.quit` имеет такой же результат, как использованный ранее `Tk.quit`).

Вот текст, который выводится на stdout при нажатии на кнопку Hello, какими бы ни были стандартные потоки для этого сценария:

```
C:\...\PP2E\Gui\Intro>python gui4.py
Hello stdout world!...
Hello stdout world!...
Hello stdout world!...
Hello stdout world!...
```

Понятие о прикреплении графических элементов к контейнерам оказывается в сердцевине всех структур в Tkinter. Однако прежде чем подробнее вникать в эту тему, обсудим небольшую деталь.

Еще раз об изменении размеров: обрезание

Ранее мы видели, как заставить графические элементы расширяться вместе с их родительским окном в результате передачи параметров expand и fill менеджеру геометрии pack. Теперь, когда у нас в окне есть несколько графических элементов, хочу раскрыть вам один из полезных секретов упаковщика. Как правило, графические элементы, упакованные первыми, обрезаются последними, когда окно уменьшается. Это означает, что порядок упаковки элементов определяет, какие из них окажутся скрытыми, если окно сделается слишком маленьким, – элементы, упакованные последними, удаляются раньше. Например, на рис. 6.14 показано, что произойдет, если окно gui4 уменьшить в интерактивном режиме.

Рис. 6.14. Уменьшение размеров gui4

Попробуйте изменить порядок строк, создающих метки и кнопки в сценарии, и посмотреть, что происходит при сжатии окна: упакованный первым всегда уходит последним. Например, если метка упаковывается последней, то, как показывает рис. 6.15, она обрезается первой, даже если прикреплена к верхнему краю: прикрепление side и порядок упаковки вместе влияют на общую структуру, но только порядок упаковки имеет значение, когда окно уменьшается.

Рис. 6.15. Label пакуется последним, обрезается первым

Это действует благодаря тому, что Tkinter внутренне запоминает порядок упаковки. Сценарии могут заранее готовиться к сжатию окон, вызывая вначале метод pack для более важных графических элементов. Например, в предстоящем обзоре Tkinter мы встретим код, который строит меню и инструментальные панели в верхней и нижней части окна. Чтобы обеспечить их исчезновение при сжатии окна в последнюю очередь, они упаковываются первыми, перед теми компонентами, которые размещаются в середине. Аналогично полосы прокрутки, содержащиеся в интерфейсах, обычно упаковываются раньше, чем прокручиваемые им элементы (например, текст или список), чтобы сохраняться при сжатии окна.

Прикрепление графических элементов к фреймам

В существенном смысле важным нововведением в этом примере является использование фреймов. Графические элементы Frame служат просто *контейнерами* для других графических элементов, благодаря чему возникает представление о GUI как об иерархиях графических элементов, или *деревьях*. Здесь в качестве охватывающего окна для трех других элементов служит win. Однако в целом, прикрепляя графические элементы к фреймам, а фреймы к другим фреймам, можно строить произвольные структуры GUI. Просто делите интерфейс пользователя на ряд прямоугольников уменьшающегося размера, реализовывайте каждый как Frame Tkinter и прикрепляйте основные графические элементы к фрейму в нужном месте экрана.

В данном сценарии в результате задания win в качестве первого аргумента в конструкторах Label и Button Tkinter прикрепляет их к Frame (они становятся дочерними для win). Сам win прикрепляется к окну верхнего уровня по умолчанию, потому что конструктору Frame не был передан родитель. Когда мы просим win выполниться (вызывая mainloop), Tkinter отображает все графические элементы в построенном нами дереве.

Три дочерних элемента позволяют теперь задать параметры pack: аргументы side говорят о том, к какой части содержащего их фрейма (то есть win) должен быть прикреплен новый графический элемент. Метка подвешивается к верхнему краю, а кнопки прикрепляются к боковым сторонам. TOP, LEFT и RIGHT являются строковыми переменными с предварительно присвоенными значениями, которые импортируются из Tkinter. Размещение графических элементов происходит несколько более тонким образом, чем просто указание стороны, но чтобы узнать почему, придется сделать краткое отступление и обсудить детали работы упаковывающего менеджера геометрии.

Порядок упаковки и прикрепление к сторонам

При отображении дерева графических элементов дочерние графические элементы появляются внутри родительских и располагаются в соответствии с порядком и параметрами упаковки. По этой причине порядок упаковки элементов не только задает порядок их обрезания, но также определяет, каким образом проявляют себя значения side в общей картине.

Вот как работает система размещения элементов упаковщиком:

1. Упаковщик начинает с *пустого* доступного пространства, в которое входит весь родительский контейнер (например, весь Frame или окно верхнего уровня).
2. Когда графический элемент упаковывается к краю, ему отдается *весь* запрашиваемый край в оставшемся пустом пространстве, и пустое пространство сокращается.
3. Последующим запросам упаковки отдается весь край того, что осталось от пустого пространства, сокращенного предшествующими запросами.
4. После того как пустое пространство отдано графическим элементам, expand делит оставшееся, а fill и anchor растягивают графические элементы и устанавливают их положение внутри выделенной им области.

Например, изменим в коде gui4 логику создания дочерних графических элементов следующим образом:

```
Button(win, text='Hello', command=greeting).pack(side=LEFT)
Label(win,  text='Hello container world').pack(side=TOP)
Button(win, text='Quit',  command=win.quit).pack(side=RIGHT)
```

В итоге получится совсем другой интерфейс, представленный на рис. 6.16, хотя в исходном файле был всего лишь перемещен на одну строку вниз код метки (сравните с рис. 6.13).

Рис. 6.16. Упаковка метки на втором месте

Теперь, несмотря на установку `side`, метка не захватывает весь верх окна, и чтобы разобраться в причине, нужно представить себе сокращающиеся пустые пространства. Так как первой пакуется кнопка Hello, ей выделяется весь левый край `Frame`. После этого метка получает весь верх того, что осталось. Наконец, кнопка Quit получает правый край остатка – прямоугольник, находящийся справа от кнопки Hello и под меткой. При сжатии этого окна графические элементы обрезаются в порядке, противоположном их упаковке: первой исчезает кнопка Quit, за ней следует метка.[1] В первоначальном варианте этого примера метка занимает весь верхний край только потому, что упаковывается первой, а не благодаря значению параметра `side`.

Снова о параметрах упаковщика expand и fill

Помимо этого, с помощью уже знакомого параметра `fill` можно растянуть графический элемент так, чтобы он занимал все пространство по выделенному ему пустому краю, а все пустое пространство, оставшееся после упаковки элементов, поровну распределялось между графическими элементами, для которых указан параметр `expand=YES`. Например, следующий код создает окно, показанное на рис. 6.17.

Рис. 6.17. Упаковка с использованием параметров expand и fill

```
Button(win, text='Hello', command=greeting).pack(side=LEFT, fill=Y)
Label(win,  text='Hello container world').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT, expand=YES, fill=X)
```

Для того чтобы все эти элементы увеличивались вместе с их окном, необходимо сделать содержащий их контейнер расширяемым: графические элементы расширяются за пределы своего первоначального положения, установленного упаковщиком, только если расширяются все их родители:

```
win = Frame()
win.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='Hello', command=greeting).pack(side=LEFT, fill=Y)
Label(win,  text='Hello container world').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT, expand=YES,fill=X)
```

При выполнении этого кода `Frame` получает весь верхний край своего родительского окна, как и раньше (то есть верхний участок корневого окна); но поскольку теперь он помечен так, что должен расширяться в неиспользуемое пространство своего родите-

[1] Технически после изменения размеров окна шаги упаковки просто выполняются заново. Но поскольку это означает, что для графических элементов, упаковываемых последними, останется недостаточно места, это равносильно тому, что элементы, упакованные последними, обрезаются первыми.

ля и заполнять его в обоих направлениях, он и все прикрепленные к нему дочерние элементы расширяются вместе с окном, как показано на рис. 6.18.

Рис. 6.18. gui4 увеличивается в расширяемом фрейме

Использование якорей вместо растягивания

Если и это не дает достаточной гибкости, то упаковщик дополнительно позволяет размещать графические элементы в отведенной для них области с помощью параметра `anchor`, а не заполнять все пространство с помощью `fill`. Параметр `anchor` принимает значение из числа констант Tkinter, указывающих восемь направлений (`N`, `NE`, `NW`, `S` и т. д.), или задающее центр (`CENTER`), например `anchor=NW` (северо-запад). Упаковщик получает при этом указание разместить графический элемент в желательном месте внутри отведенного для него пространства, если это пространство больше, чем требуется для изображения данного графического элемента.

По умолчанию якорь имеет значение `CENTER`, поэтому графические элементы выводятся в центре отведенного им пространства (выделенного им края пустого пространства), если только для них не задано положение с помощью `anchor` или они не растянуты с помощью `fill`. Чтобы продемонстрировать это, изменим код `gui4` следующим образом:

```
Button(win, text='Hello', command=greeting).pack(side=LEFT, anchor=N)
Label(win,  text='Hello container world').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT)
```

Новым здесь является только то, что кнопка Hello закреплена на северной стороне отведенного ей пространства. Так как эта кнопка была упакована первой, ей был предоставлен весь левый край родительского фрейма – больше места, чем необходимо для ее показа, поэтому по умолчанию она оказывается в середине этого края, как показано на рис. 6.16 (то есть закреплена в центре). При установке якоря в `N` она перемещается вверх по краю, как показано на рис. 6.19.

Рис. 6.19. Закрепление кнопки на севере

Помните, что параметры `fill` и `anchor` применяются *после* того, как графическому элементу выделено место у края пустого пространства с помощью параметра `side`, порядка упаковки и запроса дополнительного пространства `expand`. Путем варьирования порядка упаковки, края, заполнения и закрепления можно получить массу эффектов расположения и обрезания, и стоит потратить некоторое время, экспериментируя с альтернативными вариантами, если вы этого еще не сделали. Например, в исходном варианте этого примера метка занимает весь верхний край только потому, что была упакована первой.

Как будет показано ниже, фреймы можно вкладывать в другие фреймы, благодаря чему можно создавать сложные структуры. В действительности, поскольку каждый родительский контейнер является отдельным пустым пространством, возникает некоторый механизм обхода алгоритма выделения пустого пространства упаковщиком: чтобы лучше контролировать то, где будет отображаться некоторая группа графических элементов, просто упакуйте ее во вложенный фрейм и прикрепите этот фрейм как пакет к более крупному контейнеру. Например, ряд расположенных рядом кнопок проще разместить в собственном фрейме, чем смешивать с остальными графическими элементами.

Наконец, имейте в виду, что дерево графических элементов, создаваемое в этих примерах, в действительности является неявным: Tkinter внутренне ведет учет связей, налагаемых передаваемыми аргументами родительского графического элемента. На языке ООП это называется композицией (*composition*) – `Frame` содержит `Label` и несколько `Button`; следующим будет рассмотрен такой вид связи, как наследование (*inheritance*).

Настройка графических элементов с помощью классов

В сценариях Tkinter использовать ООП необязательно, но оно определенно может оказаться полезным. Как мы только что видели, GUI Tkinter строятся как деревья объектов экземпляров классов. Вот еще один способ применения возможностей ООП в Python к моделям GUI: задание графических элементов в результате наследования. Пример 6.18 создает окно, показанное на рис. 6.20.

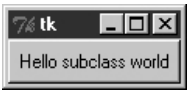


Рис. 6.20. Подкласс кнопки в действии

Пример 6.18. PP2E\Gui\Intro\gui5.py

```
from Tkinter import *

class HelloButton(Button):
    def __init__(self, parent=None, **config):          # добавить метод обратного вызова
        Button.__init__(self, parent, config)           # и упаковать себя
        self.pack()
        self.config(command=self.callback)
    def callback(self):                                 # действие по умолчанию при нажатии
        print 'Goodbye world...'                        # заменить в подклассах
        self.quit()

if __name__ == '__main__':
    HelloButton(text='Hello subclass world').mainloop()
```

В этом примере особенно смотреть не на что: он просто показывает одну кнопку, при нажатии на которую выводится сообщение и завершается программа. Но на этот раз графический элемент кнопки мы создали сами. Класс `HelloButton` наследует все, имеющееся в классе Tkinter `Button`, добавляя метод `callback` и логику в конструкторе, которая устанавливает параметр `command` равным `self.callback`, связанному методу экземпляра. При нажатии на кнопку теперь вызывается не просто функция, а *метод* `callback` нового класса графического элемента.

Здесь аргументу `**config` присваиваются ненайденные аргументы ключевых слов: они передаются конструктору `Button`. Мы уже встречались с вызовом метода графического

элемента config в конструкторе HelloButton: это просто альтернативный способ передачи параметров конфигурации после события (вместо передачи аргументов конструктора).

Какой же смысл в создании таких подклассов графических элементов? Появляется возможность конфигурировать графические элементы путем создания подклассов, а не передачи параметров. HelloButton является настоящей кнопкой: при ее создании параметры конфигурации передаются как обычно. Но можно также задать обработчик обратного вызова, переопределив в подклассе метод callback, как показано в примере 6.19.

Пример 6.19. PP2E\Gui\Intro\gui5b.py

```
from gui5 import HelloButton
class MyButton(HelloButton):        # подкласс HelloButton
    def callback(self):             # переопределить метод обработчика нажатия
        print "Ignoring press!..."
if __name__ == '__main__':
    MyButton(None, text='Hello subclass world').mainloop()
```

Вместо завершения программы при нажатии этой кнопки MyButton происходит запись в stdout и сохранение ее в верхнем положении. Вот что будет в стандартном выводе после нескольких нажатий:

```
C:\PP2ndEd\examples\PP2E\Gui\Intro>python gui5b.py
Ignoring press!...
Ignoring press!...
Ignoring press!...
Ignoring press!...
```

Как проще настраивать графические элементы – создавая подклассы или путем передачи им параметров – может быть делом вкуса. Но нужно отметить, что Tk приобретает в Python действительную объектную ориентированность, поскольку объектно-ориентированным является Python: можно задавать классы графических элементов с помощью обычной основанной на классах ОО-технологии. Следующий пример показывает еще один способ организовать специализацию.

Повторно используемые компоненты GUI и классы

Большие GUI часто строятся как подклассы Frame с реализацией обработчиков обратного вызова в виде методов. Благодаря такой структуре возникает естественное место для хранения информации между событиями: атрибуты экземпляров классов хранят *состояние*. Она позволяет также специализировать GUI, переопределяя их методы в новых подклассах, и прикреплять их к более крупным структурам GUI, чтобы повторно использовать в качестве общих компонентов. Например, текстовый редактор GUI, реализованный как подкласс Frame, можно прикреплять к любому числу других GUI и настраивать с их помощью; при правильном использовании такой текстовый редактор можно встраивать в любой интерфейс пользователя, которому требуются средства редактирования текста.

С таким текстовым редактором в виде компонента мы встретимся в главе 9. А пока проиллюстрируем идею простым примером 6.20. Сценарий *gui6.py* создает окно, показанное на рис. 6.21.

Рис. 6.21. Пользовательский Frame в действии

Пример 6.20. PP2E\Gui\Intro\gui6.py

```
from Tkinter import *

class Hello(Frame):                             # расширенный Frame
    def __init__(self, parent=None):
        Frame.__init__(self, parent)            # инициализация суперкласса
        self.pack()
        self.data = 42
        self.make_widgets()                     # прикрепить графические элементы к self
    def make_widgets(self):
        widget = Button(self, text='Hello frame world!', command=self.message)
        widget.pack(side=LEFT)
    def message(self):
        self.data = self.data + 1
        print 'Hello frame world %s!' % self.data

if __name__ == '__main__': Hello().mainloop()
```

В этом примере выводится окно с одной кнопкой. При ее нажатии запускается связанный метод `self.message`, чтобы снова произвести вывод в `stdout`. Вот вывод после четырехкратного нажатия этой кнопки; обратите внимание на сохранение своего состояния между нажатиями `self.data` (в данном случае простым счетчиком):

```
C:\...\PP2E\Gui\Intro>python gui6.py
Hello frame world 43!
Hello frame world 44!
Hello frame world 45!
Hello frame world 46!
```

Это может показаться окольным методом показа `Button` (в примерах 6.10, 6.11 и 6.12 он выполнялся меньшим числом строк). Но класс `Hello` предоставляет внешнюю организационную структуру для построения GUI. В примерах, предшествовавших предыдущему разделу, GUI создавались с помощью подхода, похожего на функции: мы вызывали конструкторы графических элементов, как если бы они были функциями, и связывали вместе графические элементы, задавая родителя при вызове конструктора графического элемента. Не было никакого представления о внешнем контексте, помимо глобальной области видимости файла модуля, содержащего вызовы графических элементов. Для простых GUI это годится, но при построении структур более крупных GUI служит причиной хрупкости кода.

При таком же создании подклассов `Frame`, которое здесь продемонстрировано, этот класс становится охватывающим контекстом GUI:

- Графические элементы добавляются путем прикрепления объектов к `self`, экземпляру подкласса контейнера `Frame` (например, `Button`).
- Обработчики обратного вызова регистрируются как связанные методы `self` и потому направляются обратно в код класса (например, `self.message`).
- Информация о состоянии сохраняется между событиями путем присвоения атрибутам `self`, видимым всеми методами обратного вызова в классе (например, `self.data`).
- Легко создать несколько экземпляров такого компонента GUI, потому что у каждого экземпляра класса отдельное пространство имен.

В некотором смысле GUI целиком становятся специализированными объектами `Frame` с расширениями, соответствующими их применению. Классы могут также предоставлять протоколы построения графических элементов (как метод `make_widgets` в данном случае), выполнять стандартные задачи конфигурирования (например, задание оп-

ций менеджера окон) и т. д. Короче, подклассы Frame дают простой способ организации совокупностей других объектов классов графических элементов.

Прикрепление классов компонентов

Более важно, вероятно, что подклассы Frame являются настоящими графическими элементами: их можно и дальше расширять и модифицировать, создавая подклассы, и прикреплять к охватывающим графическим элементам. Например, чтобы прикрепить весь пакет графических элементов, создаваемых классом, к чему-либо еще, нужно создать экземпляр класса, передав ему графический элемент, который должен стать родительским. Это иллюстрируется примером 6.21, при запуске которого создается окно, показанное на рис. 6.22.

Рис. 6.22. Прикрепленный справа компонент класса

Пример 6.21. PP2E\Gui\Intro\gui6b.py

```
from sys import exit
from Tkinter import *                    # получить классы графических элементов Tk
from gui6 import Hello                   # получить подкласс frame

parent = Frame(None)                     # создать графический элемент-контейнер
parent.pack()
Hello(parent).pack(side=RIGHT)           # прикрепить Hello, а не запускать его

Button(parent, text='Attach', command=exit).pack(side=LEFT)
parent.mainloop()
```

В этом сценарии кнопка Hello добавляется к правому краю parent – контейнера Frame. На самом деле, кнопка в правой части этого окна представляет встроенный компонент: его кнопка действительно представляет прикрепленный объект класса Python. Нажатие кнопки встроенного класса справа, как и ранее, выводит сообщение; нажатие новой кнопки закрывает GUI в результате вызова sys.exit:

```
C:\...\PP2E\Gui\Intro>python gui6b.py
Hello frame world 43!
Hello frame world 44!
Hello frame world 45!
Hello frame world 46!
```

В более сложных GUI можно прикреплять большие подклассы Frame к другим компонентам-контейнерам и разрабатывать каждый из них независимо. Пример 6.22 сам служит специализированным Frame, но прикрепляет экземпляр первоначального класса Hello более объектно-ориентированным способом. При запуске в качестве программы верхнего уровня он создает окно, идентичное показанному на рис. 6.22.

Пример 6.22. PP2E\Gui\Intro\gui6c.py

```
from Tkinter import *                    # получить классы графических элементов Tk
from gui6 import Hello                   # получить подкласс frame

class HelloContainer(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
```

```
        self.makeWidgets()
    def makeWidgets(self):
        Hello(self).pack(side=RIGHT)     # прикрепить к себе Hello
        Button(self, text='Attach', command=self.quit).pack(side=LEFT)

if __name__ == '__main__': HelloContainer().mainloop()
```

Выглядит и действует это в точности как `gui6b`, но обработчик обратного вызова добавленной кнопки регистрируется как `self.quit`, который является стандартным методом графического элемента `quit`, унаследованным от `Frame`. На этот раз окно показывает в действии два класса Python – графические элементы встроенного компонента справа (исходная кнопка Hello) и графические элементы контейнера слева.

Конечно, это простой пример (мы прикрепили здесь только одну кнопку). Но в более практических примерах набор прикрепляемых таким способом объектов классов графических элементов может быть значительно больше. Представьте себе, что вызов `Hello` заменен в этом сценарии вызовом, прикрепляющим готовый и полностью отлаженный объект калькулятора, и вам станет более понятна мощь данной парадигмы. Если кодировать все свои компоненты GUI как классы, они автоматически образуют библиотеку многократно используемых графических элементов, которые всегда можно вставлять в другие приложения.

Расширение классов компонентов

При построении GUI с помощью классов существует ряд способов повторного использования их кода в других приложениях. Чтобы расширить `Hello`, а не прикреплять его, можно переопределить некоторые его методы в новом подклассе (который сам станет специализированным графическим элементом `Frame`). Этот прием показан в примере 6.23.

Пример 6.23. PP2E\Gui\Intro\gui6d.py

```
from Tkinter import *
from gui6 import Hello

class HelloExtender(Hello):
    def make_widgets(self):                        # расширение метода
        Hello.make_widgets(self)
        Button(self, text='Extend', command=self.quit).pack(side=RIGHT)
    def message(self):
        print 'hello', self.data                   # переопределение метода

if __name__ == '__main__': HelloExtender().mainloop()
```

Метод `make_widgets` этого подкласса сначала строит графические элементы своего суперкласса, а затем добавляет справа вторую кнопку Extend, как показано на рис. 6.23.

Рис. 6.23. Модифицированные графические элементы класса в левой части

Поскольку этот подкласс переопределяет метод `message`, нажатие кнопки в левой части исходного суперкласса теперь выводит в `stdout` другую строку (когда осуществляется поиск вверх от `self`, атрибут `message` отыскивается первым в этом подклассе, а не в суперклассе):

```
C:\...\PP2E\Gui\Intro>python gui6d.py
```

```
hello 42
hello 42
hello 42
hello 42
```

Но нажатие новой правой кнопки Extend, добавленной в этот подкласс, осуществляет немедленный выход, так как метод `quit` (унаследованный от `Hello`, который наследует его от `Frame`) является обработчиком обратного вызова добавленной кнопки. В итоге этот класс модифицирует исходный, добавляя новую кнопку и изменяя поведение `message`.

Это простой пример, но он показывает технологию, оказывающуюся мощной на практике: чтобы изменить поведение GUI, можно написать новый класс, который модифицирует некоторые его части, а не изменять существующий код GUI. Главный код отлаживается только один раз, а затем модифицируется с помощью подклассов, когда возникают особые потребности.

Мораль заключается в том, что GUI, использующие Tkinter, можно кодировать без создания каких-либо новых классов, но использование классов для структурирования кода GUI в конечном итоге значительно расширяет возможности его повторного использования. При правильных действиях можно как прикреплять уже отлаженные компоненты к новым интерфейсам, так и изменять их поведение в новых внешних подклассах в соответствии с необходимыми особыми требованиями. В любом случае первоначальные затраты, связанные с использованием классов, должны в итоге окупиться в результате сокращения времени кодирования.

Автономные классы-контейнеры

Прежде чем двинуться дальше, хочу отметить, что большинство отмеченных выше выгод может быть получено в результате создания автономных классов, не являющихся производными от `Frames` или других графических элементов Tkinter. Например, класс из примера 6.24 генерирует окно, показанное на рис. 6.24.

Рис. 6.24. Упаковка автономного класса в действии

Пример 6.24. PP2E\Gui\Intro\gui7.py

```
from Tkinter import *

class HelloPackage:                             # не подкласс графических элементов
    def __init__(self, parent=None):
        self.top = Frame(parent)                # встроить во Frame
        self.top.pack()
        self.data = 0
        self.make_widgets()                     # прикрепить графические элементы к self.top
    def make_widgets(self):
        Button(self.top, text='Bye', command=self.top.quit).pack(side=LEFT)
        Button(self.top, text='Hye', command=self.message).pack(side=RIGHT)
    def message(self)):
        self.data = self.data + 1
        print 'Hello number', self.data

if __name__ == '__main__': HelloPackage().top.mainloop()
```

При выполнении этого кода кнопка Nye производит вывод в stdout, а Bye закрывает GUI и завершает работу, примерно как и раньше:

```
C:\...\PP2E\Gui\Intro>python gui7.py
Hello number 1
Hello number 2
Hello number 3
Hello number 4
```

Так же как и раньше, self.data сохраняет состояние между событиями, а обратные вызовы направляются методу self.message, имеющемуся в этом классе. Но в отличие от того, что было раньше, сам класс HelloPackage не является разновидностью графического элемента Frame. На самом деле он не является ничьей разновидностью – он служит только для создания пространств имен для хранения действительных объектов графических элементов и состояния. По этой причине графические элементы прикрепляются к self.top (встроенному фрейму Frame), а не к self. Более того, все ссылки на объект как на графический элемент должны передаваться вниз встроенному фрейму – как в вызове top.mainloop для запуска GUI.

Это приводит к тому, что приходится писать в классе несколько больше кода, но устраняет возможные конфликты имен между атрибутами, добавляемыми к self в структуре Tkinter, и существующими методами графических элементов Tkinter. Например, если вы определите в своем классе метод config, он замаскирует вызов config, экспортируемый Tkinter. Для автономных классов, как в этом примере, получены будут только те методы и атрибуты экземпляров, которые определены в вашем классе.

В действительности в Tkinter используется не очень много имен, поэтому обычно это не создает больших проблем.[1] Конечно, такая возможность существует, но, честно говоря, за восемь лет программирования на Python я не встречался с конфликтом имен Tkinter в подклассах графических элементов. Кроме того, использование автономных классов не лишено своих недостатков. Хотя в целом их можно прикреплять и создавать производные от них классы, они не вполне взаимозаменяемы с действительными объектами графических элементов. Например, вызовы настройки, выполняемые в примере 6.21 для подкласса Frame, отказывают в примере 6.25.

Пример 6.25. PP2E\Gui\Intro\gui7b.py

```
from Tkinter import *
from gui7 import HelloPackage      # или из gui7c с добавлением __getattr__

frm = Frame()
frm.pack()
Label(frm, text='hello').pack()

part = HelloPackage(frm)
part.pack(side=RIGHT)              # неудача! должно быть part.top.pack(side=RIGHT)
frm.mainloop()
```

[1] Если посмотреть исходный код модуля *Tkinter.py*, то можно заметить, что многие из создаваемых им имен атрибутов начинаются с одного символа подчеркивания, что делает их уникальными; другие имена – нет, поскольку они могут быть полезны вне реализации Tkinter (например, self.master, self.children). Странно, но Tkinter по-прежнему мало использует новый трюк Python с «псевдоприватными» атрибутами, имена которых начинаются с двух символов подчеркивания, которые влекут автоматическое добавление имени содержащего класса, благодаря чему они локализуются для создающего класса. Если Tkinter будет переписан так, чтобы использовать эту особенность, конфликты имен в подклассах графических элементов будут происходить гораздо реже.

Это будет работать не совсем правильно, потому что part не является настоящим графическим элементом. Чтобы работать с ним как с графическим элементом, нужно спуститься в part.top, прежде чем настраивать GUI, и рассчитывать на то, что имя top не будет изменено. Лучше всего осуществить это в классе, определив в нем метод, всегда направляющий обращение к неизвестным атрибутам встроенному Frame, как в примере 6.26.

Пример 6.26. PP2E\Gui\Intro\gui7c.py

```
import gui7
from Tkinter import *

class HelloPackage(gui7.HelloPackage):
    def __getattr__(self, name):
        return getattr(self.top, name)      # передать действительному графическому элементу

if __name__ == '__main__': HelloPackage().top.mainloop()
```

Но при этом требуется еще больше дополнительного кодирования в классах автономных пакетов. Как обычно, значимость всех таких компромиссов зависит от конкретного приложения.

Завершение начального обучения

В этой главе мы изучили основы программирования Python/Tkinter и познакомились с рядом простых объектов графических элементов – метками, кнопками, фреймами, а также упаковывающим менеджером геометрии. Этого достаточно для создания простых интерфейсов, но в действительности мы лишь поверхностно ознакомились с набором графических элементов Tkinter.

В следующих двух главах то, чему мы научились, будет применяться для изучения оставшейся части библиотеки Tkinter, которую мы научимся применять для создания интерфейсов, встречающихся в реальных программах GUI. В качестве предварительного обзора и предлагаемого маршрута табл. 6.1 перечисляет графические элементы, которые нам там встретятся, примерно в том порядке, в котором они будут появляться. Обратите внимание, что в этой таблице перечислены только классы графических элементов; мы также столкнемся с некоторыми дополнительными темами, относящимися к графическим элементам, которых нет в этой таблице.

Таблица 6.1. Классы графических элементов Tkinter

Класс	Описание
Label	Простая область для вывода текста
Button	Простая кнопка с меткой
Frame	Контейнер для прикрепления и размещения других графических элементов
Toplevel, Tk	Новое окно, управляемое менеджером окон
Message	Метка с несколькими строками
Entry	Простое поле для ввода текста в одну строку
Checkbutton	Кнопка с двумя состояниями, обычно используемая для выбора нескольких вариантов
Radiobutton	Кнопка с двумя состояниями, обычно используемая для выбора одного варианта

Таблица 6.1 (продолжение)

Класс	Описание
`Scale`	Ползунок со шкалой
`PhotoImage`	Объект графического изображения, используемый для вывода полноцветных картинок на других графических элементах
`BitmapImage`	Объект графического изображения, используемый для вывода растровых картинок на других графических элементах
`Menu`	Набор вариантов выбора, связанных с `Menubutton` или окном верхнего уровня
`Menubutton`	Кнопка, открывающая `Menu` с вариантами выбора или подменю
`Scrollbar`	Управляющий элемент для прокрутки других графических элементов (например, списков, холста, текста)
`Listbox`	Список для выбора имен
`Text`	Элемент для просмотра/редактирования многострочного текста, поддерживающий шрифты и т. д.
`Canvas`	Область для изображения графики с поддержкой линий, кружков, фотографий, текста и т. д.

В этой главе мы уже встречались с `Label`, `Button` и `Frame`. Чтобы облегчить усвоение оставшегося материала, он разбит на две главы: глава 7 освещает верхние элементы этой таблицы, предшествующие `Menu`, а в главе 8 представлены графические элементы, находящиеся в нижней части таблицы.

Помимо классов графических элементов, представленных в этой таблице, в библиотеке Tkinter содержатся дополнительные классы и инструменты, многие из которых также будут изучены в последующих двух главах:

Управление геометрией

`pack`, `grid`, `place`

Связанные переменные Tkinter

`StringVar`, `IntVar`, `DoubleVar`, `BooleanVar`

Составные графические элементы

`Dialog`, `ScrolledText`, `OptionMenu`

Планируемые обратные вызовы

Методы графических элементов `after`, `wait` и `update`

Прочие средства

Стандартные диалоги, буфер обмена, `bind` и `Event`, параметры настройки графических элементов, пользовательские и модальные диалоги, техника анимации

Большинство графических элементов Tkinter принадлежит к числу знакомых средств интерфейсов пользователя. Некоторые из них обладают очень богатой функциональностью. Например, класс `Text` реализует сложный графический элемент многострочного текста, поддерживающий шрифты, цвета и спецэффекты, мощности которого достаточно для реализации веб-броузера, а класс `Canvas` предоставляет множество средств графики, достаточно мощных для создания приложений обработки изображений. Кроме того, расширения Tkinter, например PMW, добавляют в инструментарий программиста GUI еще более богатые графические элементы.

Соответствие между Python/Tkinter и Tcl/Tk

В начале этой главы я упомянул, что Tkinter является интерфейсом Python к библиотеке GUI Tk, первоначально написанной для языка Tcl. В помощь читателям, переходящим на Python с Tcl и для подведения итогов некоторых основных тем, встретившихся в этой главе, в данном разделе сравниваются интерфейсы Python и Tcl к Tk. Такое сопоставление сделает также более полезными для разработчиков Python справочники по Tk, написанные для других языков.

Вообще говоря, ориентированность Tcl на командную строку сильно отличается от основанного на объектах подхода к программированию Python. Однако в отношении использования Tk синтаксические различия невелики. Вот некоторые главные отличия интерфейса Python к Tkinter:

Создание

Графические элементы создаются как объекты экземпляров классов при вызове класса графического элемента.

Владельцы (родители)

Родителями являются ранее созданные объекты, передаваемые конструкторам классов графических элементов.

Параметры графических элементов

Параметры являются аргументами, передаваемыми при помощи ключевых слов конструктору или методу `config`, либо ассоциативными массивами, проиндексированными ключевыми словами.

Операции

Операции графических элементов (действия) становятся методами объектов классов графических элементов Tkinter.

Обратные вызовы

Обработчиком обратных вызовов может быть любой вызываемый объект: функция, метод, лямбда и т. д.

Расширение

Графические элементы расширяются с помощью механизмов наследования классов Python.

Композиция

Интерфейсы строятся путем прикрепления объектов, а не конкатенации имен.

Связанные переменные (следующая глава)

Переменные, ассоциируемые с графическими элементами, являются объектами классов Tkinter с методами.

В Python команды создания графических элементов (например, `button`) являются именами классов Python, начинающимися с заглавной буквы (например, `Button`), операции с графическими элементами из двух слов (например, `add command`) становятся одним именем метода с подчеркиванием внутри (например, `add_command`), а метод «configure» можно кратко записывать «config», как в Tcl. В главе 7 будет также показано, что «переменные» Tkinter, ассоциируемые с графическими элементами, предстают в виде объектов экземпляров классов (например, `StringVar`, `IntVar`) с методами `get` и `set`, а не просто именами переменных Python или Tcl. В табл. 6.2 более конкретно приведены основные соответствия между языками.

Таблица 6.2. Соответствие между Tk и Tkinter

Операция	Tcl/Tk	Python/Tkinter
Создание	`frame .panel`	`Panel = Frame()`
Владелец	`button .panel.quit`	`Quit = Button(panel)`
Параметры	`button .panel.go -fg black`	`Go = Button(panel, fg='black')`
Настройка	`.panel.go config -bg red`	`Go.config(bg='red')` `Go['bg'] = 'red'`
Действия	`.popup invoke`	`Popup.invoke()`
Упаковка	`pack .panel -side left -fill x`	`Panel.pack(side=LEFT, fill=X)`

Конечно, некоторые из этих различий касаются не только синтаксиса. Например, Python строит внутреннее дерево объектов графических элементов исходя из аргументов, задающих родителя и передаваемых конструктору, не требуя сцепки строк путей к графическим элементам. После создания графического элемента можно непосредственно обращаться к нему по ссылке. При программировании на Tcl иногда можно спрятать записанные через точку имена путей вручную в переменных, но это не то же самое, что чисто объектно-ориентированная модель Python.

Если вы написали несколько сценариев Python/Tkinter, то отличия кодирования для объектов Python могут показаться тривиальными. В то же время, поддержка в Python ОО-технологий вносит совершенно новую составляющую в разработку для Tk: вы получаете те же самые графические элементы плюс поддержку структурирования и многократного использования кода в Python.

7

Обзор Tkinter, часть 1

«Виджеты, гаджеты и GUI... Бог мой!»

В этой главе будет продолжено рассмотрение программирования GUI в Python. В предыдущей главе рассматривались простые графические элементы, демонстрирующие основы кодирования в Python с использованием Tkinter, – кнопки, метки и тому подобное. Такое упрощение было намеренным: легче охватить взглядом картину большого GUI, когда под ногами не путаются детали интерфейса в виде графических элементов. Но теперь, после знакомства с основами, в этой и следующей главе мы переходим к представлению более сложных объектов графических элементов и средств, предоставляемых библиотекой Tkinter.

Вы увидите, как написание сценариев GUI станет и практическим, и интересным. В этих двух главах мы познакомимся с классами, участвующими в построении элементов интерфейса, которые встречаются в настоящих программах, – ползунков, флажков, меню, прокручиваемых списков, диалогов, графики и т. д. За этими главами последует еще одна, завершающая, посвященная GUI и рассматривающая более крупные GUI, в которых применяются техника и интерфейсы, показывавшиеся во всех предшествующих главах, в которых говорилось о GUI. В этих же двух главах примеры будут небольшими и самодостаточными, чтобы позволить сосредоточиться на деталях графических элементов.

Темы, освещаемые в данной главе

Формально мы уже использовали ряд простых графических элементов в главе 6 «Графические интерфейсы пользователя». Пока мы познакомились с `Label`, `Button`, `Frame` и `Tk` и попутно изучили понятия управления геометрией в `pack`. Будучи базовыми, все эти классы хорошо представляют интерфейсы Tkinter в целом и служат рабочими лошадками для типичных GUI. Например, контейнеры `Frame` служат основой иерархической структуры отображения.

В этой и следующей главах мы изучим новые параметры уже знакомых графических элементов и двинемся глубже от основ, чтобы рассказать об остальной части набора графических элементов Tkinter. Вот некоторые графические элементы и темы, которые будут изучаться в данной главе:

- Графические элементы `Toplevel` и `Tk`
- Графические элементы `Message` и `Entry`
- Графические элементы `Checkbutton`, `Radiobutton` и `Scale`
- Графика: объекты `PhotoImage` и `BitmapImage`
- Диалоги: стандартные и пользовательские

- Параметры настройки графических элементов
- Связывание событий низкого уровня
- Объекты переменных Tkinter

Глава 8 «Обзор Tkinter, часть 2» завершает краткий рассказ, представляя остальные элементы инструментария библиотеки Tkinter: меню, текст, холсты, анимацию и др.

Чтобы сделать этот обзор интереснее, я также попутно введу некоторые идеи *повторного использования компонентов*. Например, некоторые более поздние примеры будут написаны с использованием компонентов, написанных для предыдущих примеров. Хотя эти две главы знакомят с интерфейсами, основанными на графических элементах, данная книга написана о программировании на Python в целом; как будет показано, программирование с помощью Tkinter на Python может быть значительно более содержательным, чем просто вычерчивание кружков и стрелок.

Настройка внешнего вида графических элементов

До сих пор все кнопки и метки в наших примерах выводились в оформлении по умолчанию, стандартном для соответствующей платформы. В Windows это обычно означает, что они выводятся серым цветом, как в цветовой схеме, установленной на моей машине. Однако Tkinter позволяет придать графическим элементам произвольный другой внешний вид с помощью ряда параметров графических элементов и упаковщика.

Поскольку обычно я не могу устоять перед соблазном задать для графических элементов в примерах собственные настройки, хочу осветить эту тему в самом начале обзора. В примере 7.1 приведены некоторые параметры настройки, доступные в Tkinter.

Пример 7.1. PP2E\Gui\Tour\config-label.py

```
from Tkinter import *
root = Tk()
labelfont = ('times', 20, 'bold')                    # семейство, кегль, стиль
widget = Label(root, text='Hello config world')
widget.config(bg='black', fg='yellow')               # желтый текст на черной метке
widget.config(font=labelfont)                        # использовать увеличенный шрифт
widget.config(height=3, width=20)                    # первоначальный размер: линий, символов
widget.pack(expand=YES, fill=BOTH)
root.mainloop()
```

Запомните, что с помощью метода `config` графического элемента можно в любой момент переустановить его параметры, не передавая их все конструктору объекта. В данном случае мы используем этот метод для задания параметров, которые определяют окно, показанное на рис. 7.1.

Рис. 7.1. Внешний вид метки установлен пользователем

Если запустить сценарий на реальном компьютере (я, увы, не могу показать здесь в цвете, как это выглядит), то вы увидите, что текст метки выводится на черном фоне желтым цветом и шрифтом, сильно отличающимся от тех, которые мы до сих пор видели. В действительности этот сценарий настраивает метку в нескольких отношениях:

Цвет

В результате установки параметра `bg` метки ее фон устанавливается черным; аналогично параметр `fg` изменяет цвет переднего плана (текста) метки на желтый. Эти параметры цвета действуют в большинстве графических элементов Tkinter и принимают просто название цвета (например, `'blue'`) или шестнадцатеричную строку. Поддерживается большинство знакомых названий цветов (если только вам не довелось работать для Crayola[1]). Для точного указания цвета в этих параметрах можно также передавать строки с шестнадцатеричным идентификатором; они начинаются с `#` и задают цвет как насыщенность красного, зеленого и голубого, с одинаковым количеством битов в строке для каждого. Например, `'#ff0000'` задает восемь бит для каждого цвета и определяет чистый красный цвет – «f» означает в шестнадцатеричном виде четыре бита «1». Мы вернемся к этому шестнадцатеричному формату, когда встретимся с диалоговым окном выбора цвета позднее в этой главе.

Размер

Метке задается точный размер в виде количества строк в высоту и символов в ширину путем установки атрибутов `height` и `width`. Эти настройки можно применять, чтобы увеличить размер метки по сравнению с тем, который менеджер геометрии Tkinter устанавливает по умолчанию.

Шрифт

В этом сценарии задается пользовательский шрифт для текста метки путем установки для атрибута `font` значения из трех элементов, определяющих семейство шрифта, его кегль и стиль (в данном случае: Times, 20 пунктов, полужирный). Стиль шрифта может принимать значения `normal`, `bold`, `roman`, `italic`, `underline`, `overstrike` и их сочетания (например, «bold italic»). Tkinter гарантирует, что названия семейств шрифтов `Times`, `Courier` и `Helvetica` есть на всех платформах, но могут действовать и другие (например, `system` задает системный шрифт в Windows). Такие установки шрифта действуют для всех графических элементов, содержащих текст, например меток, кнопок, полей ввода, списковых окон и `Text` (последний может одновременно выводить несколько шрифтов с помощью «тегов»). Параметр `font` сохраняет возможность задавать шрифт с помощью более старых указателей в стиле X – длинных строк с дефисами и звездочками, но новый индикатор шрифта в виде набора более независим от платформы.

Структура и расширение

Наконец, метка может быть сделана расширяемой и растягиваемой в целом путем установки параметров `pack expand` и `fill`, с которыми мы познакомились в предыдущей главе: метка увеличивается вместе с окном. При максимизации этого окна черный фон заполняет весь экран, а желтый текст помещается по центру в середине – можете попробовать.

В данном сценарии итоговым результатом всех этих настроек является то, что эта метка по своему внешнему виду коренным образом отличается от тех, которые мы создавали до сих пор. Она больше не следует стандартам внешнего вида Windows, но такая согласованность не всегда важна. Tkinter предоставляет дополнительные способы настройки внешнего вида, не используемые в данном сценарии:

Граница и рельефность

Параметр графического элемента `bd=`*`N`* можно использовать для установки ширины границы, а параметр `relief=`*`S`* может задавать стиль границы; *`S`* может прини-

[1] Всемирно известный производитель товаров для детского творчества, в том числе цветных карандашей, красок, мелков, фломастеров и пр. – *Примеч. ред.*

мать значения FLAT, SUNKEN, RAISED, GROOVE, SOLID или RIDGE — все эти константы экспортирует модуль Tkinter.

Курсор

Параметр cursor можно задавать для изменения внешнего вида указателя мыши при перемещении над графическим элементом. Например, cursor='gumby' изменяет указатель на фигурку Gumby (зеленую). В число имен курсоров, часто используемых в этой книге, входят watch, pencil, cross и hand2.

Состояние

Некоторые графические элементы поддерживают понятие состояния, которое оказывает воздействие на их внешний вид. Например, параметр state=DISABLED обычно рисует на экране графический элемент закрашенным (окрашивает в серый цвет) и делает неактивным; значение NORMAL делает его обычным.

Отступы (padding)

Вокруг многих графических элементов (кнопок, меток и текста) можно добавить дополнительное пустое пространство с помощью параметров padx=*N* и pady=*N*. Интересно, что эти параметры можно задавать и в вызовах pack (тогда пустое пространство добавляется вокруг графического элемента в целом), и в самом объекте графического элемента (в результате увеличивается сам графический элемент).

Чтобы проиллюстрировать некоторые из этих дополнительных установок, пример 7.2 настраивает пользовательскую кнопку, показанную на рис. 7.2, и изменяет курсор мыши, когда он находится над кнопкой.

Рис. 7.2. Настройка кнопки в действии

Пример 7.2. PP2E\Gui\Tour\config-button.py

```
from Tkinter import *
widget = Button(text='Spam', padx=10, pady=10)
widget.pack(padx=20, pady=20)
widget.config(cursor='gumby')
widget.config(bd=8, relief=RAISED)
widget.config(bg='dark green', fg='white')
widget.config(font=('helvetica', 20, 'underline italic'))
mainloop()
```

Чтобы увидеть эффект, создаваемый в сценарии двумя этими установками, попробуйте сделать некоторые изменения на своем компьютере. Большинству графических элементов можно придать новый внешний вид таким же способом, и в этой книге мы будем неоднократно встречаться с такими параметрами. Мы встретимся также с *операционными* настройками, например focus (для получения фокуса ввода) и другими. В действительности у графических элементов могут быть десятки параметров, большинство из которых имеет разумные значения по умолчанию, создающие принятый на каждой оконной платформе внешний вид, что служит одной из причин простоты использования Tkinter. Но если требуется, Tkinter позволяет создавать значительно более индивидуальные изображения.

Окна верхнего уровня

У GUI Tkinter всегда есть *корневое* окно, которое создается по умолчанию или явно с помощью конструктора объекта `Tk`. Это главное корневое окно открывается при запуске программы и обычно служит для упаковки наиболее важных графических элементов. Помимо этого окна сценарии Tkinter могут порождать любое число независимых окон, которые создаются и всплывают по требованию в результате создания объектов графического элемента `Toplevel`.

Каждый созданный объект `Toplevel` порождает на экране новое окно и автоматически добавляет его в поток обработки цикла событий GUI программы (для активации новых окон не нужно вызывать метод `mainloop`). В примере 7.3 строится корневое окно и два всплывающих окна.

Пример 7.3. PP2E\Gui\Tour\toplevel0.py

```
import sys
from Tkinter import Toplevel, Button, Label

win1 = Toplevel()                   # два независимых окна,
win2 = Toplevel()                   # входящих в тот же процесс

Button(win1, text='Spam', command=sys.exit).pack()
Button(win2, text='SPAM', command=sys.exit).pack()

Label(text='Popups').pack()         # по умолчанию в корневое окно Tk()
win1.mainloop()
```

Сценарий `toplevel0` получает корневое окно по умолчанию (то, к которому прикрепляется `Label`, потому что для него не указан родитель), но создает также два самостоятельных окна `Toplevel`, которые появляются и действуют независимо от корневого окна, как показано на рис. 7.3.

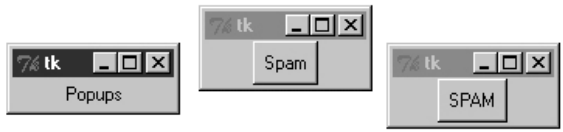


Рис. 7.3. Два окна Toplevel и корневое окно

Два окна `Toplevel` в правой части являются полноценными окнами: они могут быть независимо свернуты в значки, развернуты на весь экран и т. д. Обычно окна `Toplevel` используются для реализации многооконных интерфейсов и всплывающих модальных и немодальных диалоговых окон (подробнее о диалоговых окнах будет сказано в следующем разделе). Они сохраняются до тех пор, пока не будут явно уничтожены или создавшее их приложение не завершит работу.

Но важно знать, что хотя окна `Toplevel`s действуют независимо, они не являются независимыми процессами – если ваша программа завершится, все ее окна будут удалены, включая все окна `Toplevel`, которые она могла создать. Позднее будет показано, как обойти это правило путем запуска независимых программ GUI.

Графические элементы Toplevel и Tk

Окно `Toplevel` похоже на `Frame` тем, что отщепляется в самостоятельное окно и обладает дополнительными методами, позволяющими работать со свойствами окна верхнего уровня. Графический элемент `Tk` приблизительно похож на `Toplevel`, но используется для представления корневого окна приложения. В примере 7.3 он был получен да-

ром, потому что у Label был родитель по умолчанию; в других сценариях прямо создается корневой Tk, например:

```
root = Tk()
Label(root, text='Popups').pack()        # в явное корневое окно Tk() root window
root.mainloop()
```

В действительности, поскольку GUI Tkinter строятся в виде иерархии, по умолчанию всегда получается корневое окно, независимо от того, названо ли оно таким явно или нет. В целом корневое окно должно использоваться для вывода какого-либо рода информации верхнего уровня – если не прикрепить графические элементы к корню, при запуске сценария оно выводится как странное пустое окно. Технически, можно подавить логику создания корневого окна по умолчанию и создать несколько корневых окон с помощью графического элемента Tk, как в примере 7.4.

Пример 7.4. PP2E\Gui\Tour\toplevel1.py

```
import Tkinter
from Tkinter import Tk, Button
Tkinter.NoDefaultRoot()

win1 = Tk()          # два независимых корневых окна
win2 = Tk()

Button(win1, text='Spam', command=win1.destroy).pack()
Button(win2, text='SPAM', command=win2.destroy).pack()
win1.mainloop()
```

При запуске этого сценария выводятся два всплывающих окна, показанные на рис. 7.3 (третьего корневого окна нет). Но чаще корневой Tk используется как главное окно, а графические элементы Toplevel выводятся как всплывающие окна приложения.

Протоколы окна верхнего уровня

Графические элементы Tk и Toplevel экспортируют дополнительные методы и функции, предназначенные для той роли, которую они играют на верхнем уровне, что иллюстрируется примером 7.5.

Пример 7.5. PP2E\Gui\Tour\toplevel2.py

```
##############################################################################
# Всплывают три новых окна со стилями, destroy() убивает одно окно, quit() убивает все окна
# и приложение; у окон верхнего уровня есть заголовок, значок (в Unix), превращение в значок
# и обратно и протокол для событий wm; всегда есть корневое окно приложения, создаваемое
# по умолчанию или как явный объект Tk(); все окна верхнего уровня являются контейнерами,
# и никогда не упаковываются и не помещаются в сетку;
# Toplevel похож на frame, но это новое окно, которое может содержать меню;
##############################################################################

from Tkinter import *
root = Tk()                                                  # явное создание root

trees = [('The Larch!',          'light blue'),
         ('The Pine!',           'light green'),
         ('The Giant Redwood!',  'red')]

for (tree, color) in trees:
    win = Toplevel(root)                                     # новое окно
    win.title('Sing...')                                     # установить заголовок
    win.protocol('WM_DELETE_WINDOW', lambda:0)               # игнорировать закрытие
```

```
    msg = Button(win, text=tree, command=win.destroy)          # убивает одно окно
    msg.pack(expand=YES, fill=BOTH)
    msg.config(padx=10, pady=10, bd=10, relief=RAISED)
    msg.config(bg='black', fg=color, font=('times', 30, 'bold italic'))

root.title('Lumberjack demo')
Label(root, text='Main window', width=30).pack()
Button(root, text='Quit All', command=root.quit).pack()        # убивает все приложение
root.mainloop()
```

Эта программа добавляет графические элементы в корневое окно `Tk`, сразу выводит три окна `Toplevel` с прикрепленными кнопками и использует специальные протоколы верхнего уровня. При выполнении создается картинка, переданная в черно-белом изображении на рис. 7.4 (на цветном мониторе текст кнопок синего, зеленого и красного цвета).

Рис. 7.4. Три настроенных окна Toplevel

Здесь следует отметить несколько деталей, касающихся функционирования, которые станут более заметными, если вы запустите сценарий на своей машине:

Перехват закрытия: protocol

Так как этот сценарий перехватывает событие закрытия окна менеджера окон с помощью метода графического элемента верхнего уровня `protocol`, при нажатии X в правом верхнем углу какого-либо из трех всплывающих окон `Toplevel` ничего не происходит. Строка с именем `WM_DELETE_WINDOW` обозначает операцию закрытия. С помощью этого интерфейса можно запретить закрытие окон, кроме как посредством создаваемых в сценарии графических элементов – создаваемая этим сценарием функция `lambda:0` лишь возвращает ноль и больше ничего не делает.

Закрытие одного окна: destroy

При нажатии на большую черную кнопку в любом из трех всплывающих окон уничтожается только это окно, потому что оно выполняет для графического элемента метод `destroy`. Остальные окна продолжают существовать, как свойственно всплывающим диалоговым окнам.

Закрытие всех окон: quit

Чтобы закрыть сразу все окна и завершить приложение GUI (в действительности его активный вызов `mainloop`), кнопка корневого окна выполняет метод `quit`. Нажатие кнопки в корневом окне заканчивает выполнение приложения.

Заголовки окон: title

В главе 6 говорилось о методе `title` графических элементов окон верхнего уровня (`Tk` и `Toplevel`), позволяющем изменять текст, выводимый на верхней кромке окна. В данном случае текст заголовка окна установлен как строка «Sing...», замещающая «tk» – текст по умолчанию.

Управление геометрией

Окна верхнего уровня служат контейнерами для других графических элементов, подобно отдельному `Frame`. Однако в отличие от фреймов, графические элементы – окна верхнего уровня – сами не упаковываются (и не размещаются в сетке другим менеджером геометрии). Для встраивания графических элементов этот сценарий передает свои окна в аргументах, задающих родительское окно, конструкторам меток и кнопок.

Кроме того, графические элементы окон верхнего уровня поддерживают другие типы протоколов, которые будут позднее использованы в данном обзоре:

Состояние

Методы графических элементов верхнего уровня `iconify` и `withdraw` позволяют сценариям прятать или удалять окна на лету; `deiconify` перерисовывает скрытое или удаленное окно. Метод `state` запрашивает состояние окна (он возвращает значение «iconic», «withdrawn» или «normal»), а `lift` и `lower` поднимают или опускают окно относительно других. Их использование демонстрируется в сценариях будильника в конце главы 8.

Меню

Каждое окно верхнего уровня может иметь собственное меню; его горизонтальная строка связывается с выпадающим списком опций с помощью параметра `menu` элементов `Tk` и `Toplevel`. Эта панель меню выглядит соответствующим образом на каждой платформе, где выполняется сценарий. Меню будут изучаться в начале главы 8.

Обратите внимание, что в этом сценарии при вызове конструктора `Toplevel` ему явным образом передается родительский графический элемент – корневое окно `Tk` (то есть `Toplevel(root)`). Окна `Toplevel` можно связывать с родительскими, как другие графические элементы, хотя зрительно они не встраиваются в родительские окна. Такой способ написания сценария имел целью избежать одной странной особенности. Допустим, код написан так:

```
win = Toplevel()                                    # новое окно
```

Тогда, если корневого окна `Tk` еще не существует, этот вызов создаст корневое окно `Tk` по умолчанию, которое станет родителем `Toplevel`, как при всяком другом вызове графического элемента без задания родителя. Проблема в том, что критическим становится расположение следующей строки:

```
root = Tk()                                         # явное создание корня
```

Если поместить эту строку *выше* вызовов `Toplevel`, она создаст одно корневое окно, как и предполагается. Но если поставить эту строку *ниже* вызовов `Toplevel`, то Tkinter создаст корневое окно `Tk`, которое будет отлично от созданного сценарием при явном вызове `Tk`. Это приведет к созданию двух корневых окон `Tk`, как в примере 7.5. Переместите вызов `Tk` под вызовы `Toplevel`, перезапустите сценарий, и вы увидите, что я имею в виду – вы получите четвертое, совершенно пустое окно! Чтобы избежать таких странностей, возьмите за правило создавать корневые окна `Tk` в начале сценариев и явным образом.

Все интерфейсы протоколов верхнего уровня доступны только в графических элементах окон верхнего уровня, но часто доступ к ним можно получить через атрибут гра-

фического элемента master, представляющий собой ссылку на родительское окно графического элемента. Например, чтобы изменить заголовок окна, в котором содержится фрейм, можно сказать что-то вроде следующего:

```
theframe.master.title('Spam demo')    # master является окном-контейнером
```

Естественно, делать так можно только при уверенности, что фрейм будет использован только в одном типе окна. Например, прикрепляемые компоненты общего назначения, кодируемые в виде классов, должны оставить установку свойств окон своим приложениям-клиентам.

Для графических элементов верхнего уровня существуют другие инструменты, некоторые из которых могут не встретиться в этой книге. Например, в менеджерах окон Unix можно вызывать относящиеся к значкам методы для изменения растровых изображений, используемых в окнах верхнего уровня (iconbitmask), и установки имени для значка окна (iconname). Поскольку эти параметры значков можно применять только в сценариях, выполняемых под Unix, подробности, касающиеся этой темы, смотрите в других ресурсах по Tk и Tkinter. А сейчас нас ожидает следующая запланированная остановка в нашей экскурсии, где будет рассказано об одном из наиболее частых применений окон верхнего уровня.

Диалоги

Диалоги – это окна, выводимые сценарием для показа или запроса дополнительной информации. Есть два вида диалогов: модальные и немодальные:

- *Модальные* диалоги блокируют остальную часть интерфейса, пока окно диалога не будет освобождено; выполнение программы будет продолжено после получения диалогом ответа пользователя.
- *Немодальные* диалоги могут оставаться на экране неопределенное время, не создавая при этом помех другим окнам интерфейса; обычно они в любой момент могут принимать входные данные.

Независимо от модальности, диалоги обычно реализуются с помощью объекта окна Toplevel, с которым мы познакомились в предыдущем разделе, создаете вы Toplevel или нет. Есть, в сущности, три способа представить пользователю всплывающий диалог с помощью Tkinter: с помощью вызовов стандартных диалогов, при использовании современного объекта Dialog и путем создания пользовательских диалоговых окон с помощью Toplevel и других типов графических элементов. Рассмотрим основы использования всех трех схем.

Стандартные диалоги

Вызовы стандартных диалогов проще, поэтому начнем с них. С Tkinter поступает набор готовых диалоговых окон, реализующих многие из наиболее часто встречающихся всплывающих окон, генерируемых программами, – диалоги выбора файла, всплывающие окна ошибок и предупреждений и окна вопросов и ответов. Они называются стандартными («standart», а иногда «common») диалогами, поскольку входят в состав библиотеки Tkinter и используют библиотечные вызовы для конкретных платформ, чтобы принять вид, свойственный данной платформе. Например, диалог открытия файла Tkinter выглядит как любой другой такой диалог в Windows.

Все стандартные диалоги являются модальными (возврата из них не происходит, пока пользователь не закроет диалоговое окно) и при своем показе блокируют главное окно программы. Сценарии могут настраивать окна этих диалогов, передавая текст

сообщения, заголовки и тому подобное. Поскольку их весьма просто использовать, сразу рассмотрим следующий пример:

Пример 7.6. PP2E\Gui\Tour\dlg1.pyw

```
from Tkinter import *
from tkMessageBox import *

def callback():
    if askyesno('Verify', 'Do you really want to quit?'):
        showwarning('Yes', 'Quit not yet implemented')
    else:
        showinfo('No', 'Quit has been cancelled')

errmsg = 'Sorry, no Spam allowed!'
Button(text='Quit', command=callback).pack(fill=X)
Button(text='Spam', command=(lambda: showerror('Spam', errmsg))).pack(fill=X)
mainloop()
```

В качестве оболочки вызова `showerror` здесь использована анонимная лямбда-функция, передающая последнему два зашитых в коде аргумента (вспомните, что обратные вызовы нажатия кнопки не получают аргументов от самого Tkinter). При выполнении этот сценарий создает главное окно, показанное на рис. 7.5.

Рис. 7.5. Главное окно dlg1: кнопки для показа всплывающих окон

Нажатие кнопки Quit в этом окне показывает диалог `askyesno` (рис. 7.6) путем вызова стандартной функции `askyesno` из модуля `tkmessagebox`. В Unix и Macintosh это выглядит иначе, но в Windows именно так. Данный диалог блокирует программу, пока пользователь не щелкнет по одной из кнопок; при выборе кнопки Yes (или нажатии клавиши Enter) вызов диалога возвращает значение «истина», и сценарий показывает стандартный диалог `showwarning` (рис. 7.6), вызывая `showwarning`.

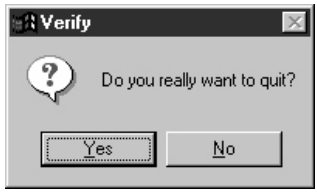


Рис. 7.6. Диалоги askyesno и showwarning сценария dlg1 (в Windows)

В диалоге `showwarning` пользователь может только нажать OK. Если щелкнуть No в диалоге `askyesno`, то вызов `showinfo` создает соответствующее всплывающее окно (рис. 7.7). Наконец, если в главном окне щелкнуть по кнопке Spam, то с помощью стандартного вызова `showerror` создается стандартный диалог `showerror` (рис. 7.7).

Конечно, в результате создается множество всплывающих окон, и не следует злоупотреблять этими диалогами (обычно лучше применять окна с полями ввода, остающиеся на экране длительное время, а не отвлекать пользователя всплывающими окнами). Но в нужных случаях такие всплывающие диалоги сокращают время кодирования и обеспечивают привычный внешний вид.

Рис. 7.7. Диалоги showinfo и showerror сценария dlg1

«Умная» и многократно используемая кнопка quit

Для некоторых из этих готовых диалогов можно найти лучшее применение. В примере 7.7 реализована прикрепляемая кнопка Quit, которая с помощью стандартных диалогов получает подтверждение запроса на завершение. Поскольку это класс, его можно прикреплять и повторно использовать в любом приложении, где требуется кнопка Quit с запросом подтверждения. Так как в этой кнопке использованы стандартные диалоги, она должным образом выглядит на любой платформе GUI.

Пример 7.7. PP2E\Gui\Tour\quitter.py

```
##############################################
# кнопка quit, которая проверяет запрос на выход;
# для повторного использования прикрепите экземпляр к другому gui
##############################################

from Tkinter import *                           # получить классы графических элементов
from tkMessageBox import askokcancel            # получить готовый стандартный диалог

class Quitter(Frame):                           # подкласс нашего GUI
    def __init__(self, parent=None):            # метод конструктора
        Frame.__init__(self, parent)
        self.pack()
        widget = Button(self, text='Quit', command=self.quit)
        widget.pack(side=LEFT)
    def quit(self):
        ans = askokcancel('Verify exit', "Really quit?")
        if ans: Frame.quit(self)

if __name__ == '__main__':  Quitter().mainloop()
```

Этот модуль предназначен для использования в других программах, но может запускаться самостоятельно и тогда выводит кнопку, которая в нем реализована. На рис. 7.8 слева вверху показана сама кнопка Quit и диалог `askokcancel` запроса подтверждения, выведенный при нажатии Quit.

Если нажать OK в этом окне, то `Quitter` выполнит метод quit элемента `Frame` и закроет GUI, к которому прикреплена кнопка (на самом деле завершит вызов `mainloop`). Но

Рис. 7.8. Модуль Quitter с диалогом askokcancel

чтобы действительно оценить пользу, приносимую такими подпружиненными кнопками, изучим GUI клиента, приведенный в следующем разделе.

Панель запуска демонстрации диалогов

Пока мы увидели лишь несколько стандартных диалогов, но их число значительно больше. Не станем показывать их на серых экранных снимках, а напишем на Python демонстрационный сценарий, который может генерировать их по требованию. Вот один из способов сделать это. Во-первых, напишем модуль, приведенный в примере 7.8, который определяет таблицу соответствия между именем демонстрационной программы и вызовом стандартного диалога (будем использовать лямбду для оборачивания вызова, если функции диалога нужно передать дополнительные аргументы).

Пример 7.8. PP2E\Gui\Tour\dialogTable.py

```
# определяет таблицу demos, содержащую имя:обратный вызов

from tkFileDialog   import askopenfilename        # получить стандартные диалоги
from tkColorChooser import askcolor               # они находятся в Lib/lib-tk
from tkMessageBox   import askquestion, showerror
from tkSimpleDialog import askfloat

demos = {
    'Open':  askopenfilename,
    'Color': askcolor,
    'Query': lambda: askquestion('Warning', 'You typed "rm *"\nConfirm?'),
    'Error': lambda: showerror('Error!', "He's dead, Jim"),
    'Input': lambda: askfloat('Entry', 'Enter credit card number')
}
```

Я поместил эту таблицу в модуль, чтобы использовать ее в качестве основы будущих демонстрационных сценариев (работать с диалогами веселее, чем выводить в `stdout`). Затем мы напишем сценарий Python из примера 7.9, который просто генерирует кнопки для всех этих элементов таблицы – использует ее ключи как метки кнопок, а значения как обработчики обратного вызова для кнопок.

Пример 7.9. PP2E\Gui\Tour\demoDlg.py

```
from Tkinter import *              # получить базовый набор графических элементов
from dialogTable import demos      # обработчики обратного вызова кнопок
from quitter import Quitter        # прикрепить к себе объект Quitter

class Demo(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        Label(self, text="Basic demos").pack()
        for (key, value) in demos.items():
            Button(self, text=key, command=value).pack(side=TOP, fill=BOTH)
        Quitter(self).pack(side=TOP, fill=BOTH)

if __name__ == '__main__': Demo().mainloop()
```

Этот сценарий при выполнении в качестве самостоятельной программы создает окно, показанное на рис. 7.9: это панель демонстрационных кнопок, которые при нажатии просто маршрутизируют управление обратно по значениям таблицы из модуля `dialogTable`.

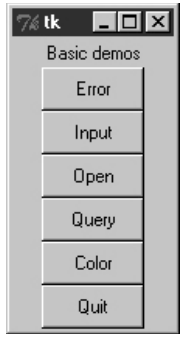


Рис. 7.9. Главное окно demoDlg

Обратите внимание, что поскольку этот сценарий управляется содержимым словаря из модуля `dialogTable`, можно регулировать набор выводимых демонстрационных кнопок, изменяя только `dialogTable` (никакого исполняемого кода в `demoDlg` менять не нужно). Отметьте также, что кнопка Quit является в данном случае прикрепленным экземпляром класса `Quitter` из предыдущего раздела – по крайней мере эту часть кода уже не нужно будет писать снова.

Мы уже видели некоторые диалоги, запускаемые другими кнопками окна этой демонстрационной панели, поэтому я коснусь здесь только новых. Например, нажатие в главном окне кнопки Query генерирует стандартный диалог, показанный на рис. 7.10.

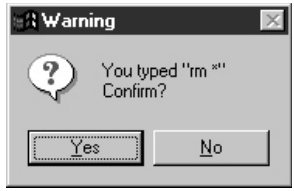


Рис. 7.10. Запрос demoDlg, диалог askquestion

Этот диалог `askquestion` выглядит как `askyesno`, который мы видели раньше, но в действительности возвращает строку «yes» или «no» (`askyesno` и `askokcancel` возвращают 1 или 0, «истину» или «ложь»). Нажатие кнопки демонстрационной панели Input генерирует стандартное диалоговое окно `askfloat`, показанное на рис. 7.11.

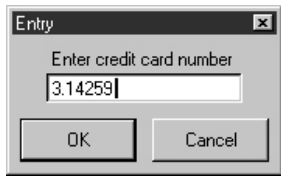


Рис. 7.11. Ввод demoDlg, диалог askfloat

Этот диалог автоматически проверяет введенные данные на соответствие синтаксису числа с плавающей точкой, прежде чем возвратиться, и является представителем группы диалоговых окон для ввода одного значения (помимо `askinteger` и `askstring`, предлагающих ввести целое число и строку). Он возвращает введенные данные как объект числа с плавающей точкой (а не строку) при нажатии кнопки OK и клавиши

Enter либо объект Python None, если пользователь щелкнет по Cancel. Два родственных ему диалога возвращают объекты целого числа и строки.

При нажатии на демонстрационной панели кнопки Open мы получаем стандартный диалог открытия файла, создаваемый в результате вызова askopenfilename и показанный на рис. 7.12. Это внешний вид в Windows; в Linux диалог выглядит совершенно иначе, как и должно быть.

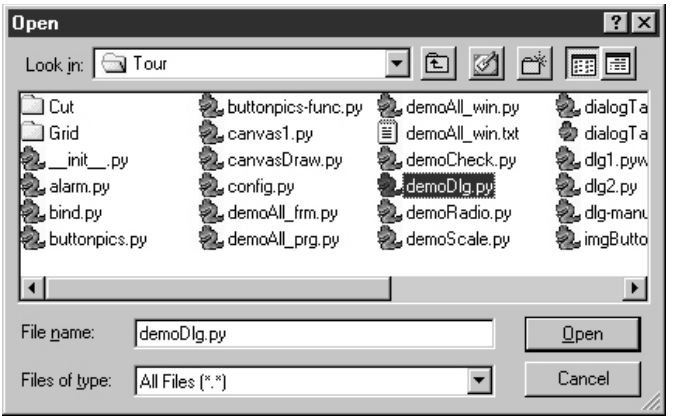


Рис. 7.12. Открытие файла в demoDlg, диалог askopenfilename dialog

Аналогичный диалог для выбора имени сохраняемого файла (save-as) создается при вызове asksaveasfilename (пример можно найти в разделе, посвященном графическому элементу Text, в главе 8). Оба файловых диалога дают пользователю возможность перемещаться по файловой системе для выбора нужного имени файла, которое возвращается вместе с полным путем каталога при нажатии кнопки Open; если нажата Cancel, возвращается пустая строка. Оба диалога поддерживают дополнительные протоколы, не показанные в этом примере:

- Им можно передать аргумент ключевого слова filetypes – группа шаблонов имен для выбора файлов, появляющихся в выпадающем списке «Files of type» в нижней части диалога.
- Им можно передать initialdir (начальный каталог), initialfile (для «File name»), title (заголовок окна диалога), defaultextension (расширение, добавляемое, когда у выбранного файла расширения нет) и parent (для отображения в виде встроенного дочернего элемента, а не всплывающего диалога).
- Можно заставить их запомнить последний выбранный каталог путем использования экспортированных объектов вместо этих вызовов функций.

Большинство из этих интерфейсов позднее будут использованы в книге, особенно в файловых диалогах примера PyEdit в главе 9 «Более крупные примеры GUI» (но можно забежать вперед сейчас, чтобы узнать дополнительные подробности). Наконец, кнопка Color демонстрационной панели запускает стандартный вызов askcolor, который генерирует стандартный диалог выбора цвета, показанный на рис. 7.13.

При нажатии в нем кнопки OK возвращается структура данных, идентифицирующая выбранный цвет, которую можно использовать в любом контексте цвета в Tkinter. В нее входят значения RGB и шестнадцатеричная строка цветов (например, ((160, 160, 160), '#a0a0a0')). Подробнее об использовании этого набора будет сказано несколько позже. При нажатии Cancel сценарий возвращает набор, состоящий из двух None.

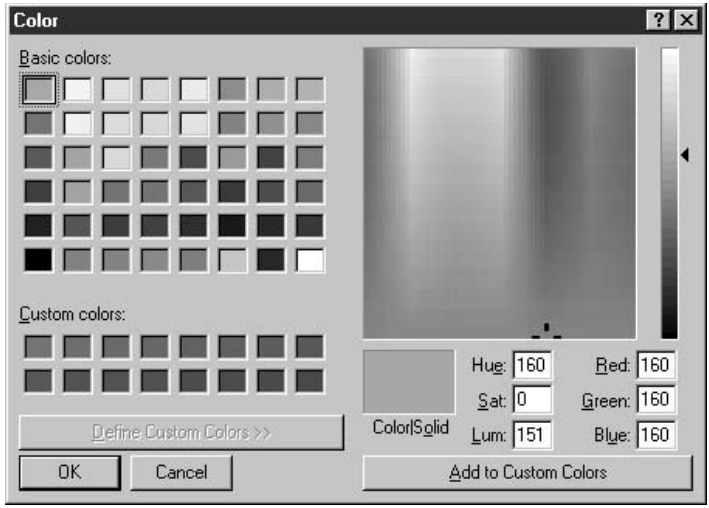


Рис. 7.13. Выбор цвета в demoDlg, диалог askcolor

Вывод результатов, возвращаемых диалогами (и передача данных обратного вызова с помощью лямбды)

Демонстрационная панель запуска диалогов выводит стандартные диалоги и может быть использована для вывода других диалогов путем изменения импортируемого модуля `dialogTable`. Однако в существующем виде этот код лишь *показывает* диалоги; неплохо было бы посмотреть на возвращаемые ими значения, чтобы знать, как использовать их в сценариях. В примере 7.10 добавлен вывод результатов стандартных диалогов в стандартный выходной поток `stdout`.

Пример 7.10. PP2E\Gui\Tour\demoDlg-print.py

```
##############################################################################################
# То же, но после показа возвращается значение вызова диалога; лямбда сохраняет данные
# из локальной области для передачи обработчику (обработчики кнопок обычно не получают
# аргументов) и работает подобно такой команде def: def func(self=self, name=key):
# self.printit(name)
##############################################################################################

from Tkinter import *                # получить базовый набор графических элементов
from dialogTable import demos        # обработчики обратного вызова кнопок
from quitter import Quitter          # прикрепить к себе объект Quitter
class Demo(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        Label(self, text="Basic demos").pack()
        for (key, value) in demos.items():
            func = (lambda self=self, name=key: self.printit(name))
            Button(self, text=key, command=func).pack(side=TOP, fill=BOTH)
        Quitter(self).pack(side=TOP, fill=BOTH)
    def printit(self, name):
        print name, 'returns =>', demos[name]()      # получение, вызов, вывод

if __name__ == '__main__': Demo().mainloop()
```

Этот сценарий строит то же самое главное окно панели кнопок, но обратите внимание, что обработчик обратного вызова теперь является анонимной функцией, созданной с помощью лямбды, а не прямой ссылкой на вызов диалога в импортированном словаре `dialogTable`:

```
func = (lambda self=self, name=key: self.printit(name))
```

Мы фактически впервые использовали лямбду подобным образом, поэтому разберемся в том, что происходит. Так как обратные вызовы нажатий кнопок выполняются без аргументов, то при необходимости передать обработчику дополнительные данные для него нужно создать оболочку в виде объекта, который запоминает эти дополнительные данные и передает их. В данном случае при нажатии кнопки выполняется функция, создаваемая лямбдой, – слой косвенного вызова, сохраняющий информацию из окружающей области путем присвоения ее аргументам по умолчанию. Достигаемый эффект состоит в том, что реальный обработчик, `printit`, получает дополнительный аргумент `name`, давая демонстрацию, связанную с нажатой кнопкой, несмотря на то, что этот аргумент не был передан самим Tkinter.

Заметьте, однако, что эта лямбда присваивает `self` и `key` значениям по умолчанию, чтобы сохранить их для использования в обратных вызовах. Как и все функции, результаты лямбды имеют доступ только к своей локальной области видимости, охватывающей глобальной области видимости модуля и области видимости встроенных имен, но не локальной области видимости функции метода, создавшей их, и именно там находится в действительности имя `self`. Так как *связанные методы* помнят и объект `self`, и функцию метода, эту лямбду можно написать и так:

```
func = (lambda handler=self.printit, name=key: handler(name))
```

Здесь можно также использовать вызываемый объект класса, который сохраняет состояние в виде атрибутов экземпляра (смотрите подсказку в учебном примере `__call__` главы 6). Но практический совет заключается в следующем: если нужно, чтобы результат лямбды в последующих вызовах использовал имена из охватывающей области, просто передайте их в качестве значений по умолчанию.

При выполнении этого сценария выводятся значения, возвращаемые диалогами; вот вывод, полученный после нажатия всех кнопок демонстраций в главном окне и выбора в каждом диалоге обеих кнопок Cancel/No и OK/Yes:

```
C:\...\PP2E\Gui\Tour>python demoDlg-print.py
Error returns => ok
Input returns => None
Input returns => 3.14159
Open returns =>
Open returns => C:/PP2ndEd/examples/PP2E/Gui/Tour/demoDlg-print.py
Query returns => no
Query returns => yes
Color returns => (None, None)
Color returns => ((160, 160, 160), '#a0a0a0')
```

Показав результаты всех диалоговых окон, я хочу продемонстрировать фактическое использование одного из них.

Предоставление пользователям возможности динамического выбора цвета

Стандартный диалог для выбора цвета не просто украшение – сценарии могут передавать возвращаемую этим диалогом шестнадцатеричную строку цветов в уже знакомые нам параметры `bg` и `fg` настройки цветов графических элементов. Это значит, что `bg` и `fg` принимают имя цвета (например, «blue») и результирующие строки `askcolor`,

которые начинаются с # (например, #a0a0a0 из последней строки вывода в предыдущем разделе).

Это добавляет новое измерение в модификацию GUI Tkinter: вместо жесткой прошивки цветов в создаваемых GUI можно сделать кнопку, выводящую диалоговые окна для выбора цвета, с помощью которых пользователи могут осуществлять настройку цветов на лету. Нужно просто передать строку цвета методам config в обработчиках обратного вызова, как в примере 7.11.

Пример 7.11. PP2E\Gui\Tour\setcolor.py

```
from Tkinter import *
from tkColorChooser import askcolor

def setBgColor():
    (triple, hexstr) = askcolor()
    if hexstr:
        print hexstr
        push.config(bg=hexstr)

root = Tk()
push = Button(root, text='Set Background Color', command=setBgColor)
push.config(height=3, font=('times', 20, 'bold'))
push.pack(expand=YES, fill=BOTH)
root.mainloop()
```

Этот сценарий создает при запуске окно, показанное на рис. 7.14 (фон его кнопки зеленоватый, и вам придется поверить мне на слово). Нажатие кнопки выводит диалог выбора цвета, показывавшийся ранее; цвет, выбранный в этом окне, становится цветом фона этой кнопки после нажатия OK.

Рис. 7.14. Главное окно setcolor

Строки цветов выводятся также в поток stdout (окно консоли). Запустите этот сценарий на своем компьютере и поэкспериментируйте с возможными настройками цветов:

```
C:\...\PP2E\Gui\Tour>python setcolor.py
#c27cc5
#5fe28c
#69d8cd
```

Другие вызовы стандартных диалогов

Мы уже видели большую часть стандартных диалогов и будем пользоваться ими в примерах на протяжении оставшейся части книги. Если нужны дополнительные сведения о других имеющихся вызовах и параметрах, обратитесь к другой документации Tkinter или просмотрите исходный код модулей, используемых в начале модуля dialogTable; все они являются обычными файлами Python, установленными на вашей машине в подкаталоге *lib-tk* стандартной библиотеки Python. И сохраните этот пример с демонстрационной панелью на будущее: мы снова воспользуемся им позднее, когда встретимся с другими графическими элементами, похожими на кнопки.

Модуль Dialog для окон в старом стиле

В более старом коде Python можно иногда увидеть диалоги, кодируемые с помощью стандартного модуля Dialog. Сейчас он несколько устарел и использует внешний вид, характерный для X Windows; но на случай, если вам придется встретить такой код при сопровождении программ Python, пример 7.12 дает представление об этом интерфейсе.

Пример 7.12. PP2E\Gui\Tour\dlg-old.py

```
from Tkinter import *
from Dialog import Dialog

class OldDialogDemo(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        Pack.config(self)  # same as self.pack()
        Button(self, text='Pop1', command=self.dialog1).pack()
        Button(self, text='Pop2', command=self.dialog2).pack()
    def dialog1(self):
        ans = Dialog(self,
                     title   = 'Popup Fun!',
                     text    = 'An example of a popup-dialog '
                               'box, using older "Dialog.py".',
                     bitmap  = 'questhead',
                     default = 0, strings = ('Yes', 'No', 'Cancel'))
        if ans.num == 0: self.dialog2()
    def dialog2(self):
        Dialog(self, title   = 'HAL-9000',
                     text    = "I'm afraid I can't let you do that, Dave...",
                     bitmap  = 'hourglass',
                     default = 0, strings = ('spam', 'SPAM'))

if __name__ == '__main__': OldDialogDemo().mainloop()
```

В Dialog передается кортеж наименований кнопок и сообщение, а обратно возвращается индекс нажатой кнопки (индекс самой левой – ноль). Окна Dialog модальные: остальные окна приложения отключены, когда Dialog ждет ответа пользователя. При нажатии кнопки Pop2 в главном окне, создаваемом этим сценарием, выводится второй диалог, как показано на рис. 7.15.

Рис. 7.15. Диалог в старом стиле

Этот сценарий выполняется в Windows, и, как можно видеть, нисколько не похож на тот диалог, который принят на этой платформе для задания вопроса. В действительности при вызове на любой платформе этот диалог имеет внешний вид, принятый в X Windows. Из-за внешнего вида Dialog и дополнительной сложности его программирования лучше использовать вызовы стандартных диалогов, приведенных в предыдущем разделе.

Пользовательские диалоги

Для всех встреченных до сих пор диалогов внешний вид и способы взаимодействия стандартны. Для многих задач этого достаточно, но иногда требуются несколько более специфические диалоги. Например, формы, требующие заполнения нескольких полей (скажем, ввода имени, возраста и размера ботинок), не обеспечиваются непосредственно библиотекой стандартных диалогов. Можно было бы поочередно показывать диалоговые окна для ввода каждого запрашиваемого поля, но такой интерфейс нельзя назвать дружественным пользователю.

Пользовательские диалоги поддерживают произвольные интерфейсы, но и программировать их сложнее всего. Впрочем, многого для этого не требуется: создать всплывающее окно как `Toplevel` с прикрепленными графическими элементами и устроить обработчик обратного вызова, собирающий данные, введенные пользователем (если они есть), и удаляющий окно. Чтобы сделать такой пользовательский диалог модальным, необходимо дожидаться ответа, для чего окну передается фокус ввода, другие окна делаются неактивными и происходит ожидание события. Это иллюстрируется примером 7.13.

Пример 7.13. PP2E\Gui\Tour\dlg-custom.py

```
import sys
from Tkinter import *
makemodal = (len(sys.argv) > 1)

def dialog():
    win = Toplevel()                                  # создать новое окно
    Label(win, text='Hard drive reformatted!').pack()  # добавить несколько
                                                      # графических элементов
    Button(win, text='OK', command=win.destroy).pack()  # обратный вызов разрушает окно
    if makemodal:
        win.focus_set()          # перехватить фокус ввода,
        win.grab_set()           # отключить другие окна,
        win.wait_window()        # ждать удаления win
    print 'dialog exit'          # или сразу вернуться

root = Tk()
Button(root, text='popup', command=dialog).pack()
root.mainloop()
```

Этот сценарий создает всплывающее окно в модальном или немодальном режиме в зависимости от глобальной переменной `makemodal`. При выполнении без аргументов командной строки выбирается немодальный стиль, как на рис. 7.16.

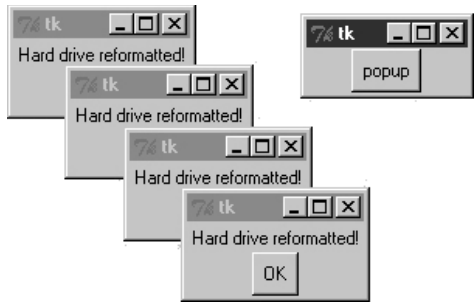


Рис. 7.16. Немодальные пользовательские диалоги в действии

Здесь окно в правом верхнем углу – корневое; при нажатии в нем кнопки «popup» создается новое всплывающее диалоговое окно. Поскольку в этом режиме диалоги не являются модальными, корневое окно сохраняет активность после показа диалога. В действительности немодальные диалоги не блокируют другие окна, поэтому можно повторно нажимать кнопку в корневом окне, создавая столько копий всплывающего окна, сколько поместится на экране. Любые из этих окон можно убить нажатием их кнопок OK, при этом остальные окна сохраняются на экране.

Создание модальных пользовательских диалогов

Если запустить сценарий, задав в командной строке аргумент (например, `python dlg-custom.py 1`), всплывающее окно будет сделано модальным. Так как модальные диалоги сосредоточивают на себе все внимание интерфейса, главное окно становится неактивным до того момента, когда будет убито всплывающее окно: пока диалог открыт, нельзя даже щелкнуть по корневому окну, чтобы активизировать его. Поэтому создать на экране больше одного всплывающего окна нельзя, как это показано на рис. 7.17.

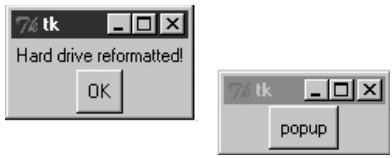


Рис. 7.17. Модальный пользовательский диалог d действии

На практике возврата из функции `dialog` в этом сценарии не происходит, пока диалоговое окно в левой части не будет закрыто нажатием кнопки OK. В результате модальные диалоги накладывают на модель программирования, в других отношениях управляемую событиями, модель вызова функций – введенные пользователем данные можно обрабатывать сразу, а не в обработчике обратного вызова, запускаемом в какой-то неопределенный будущий момент времени.

Однако навязывание такой линейной логики управления в GUI требует некоторой дополнительной работы. Секрет блокирования других окон и ожидания ответа сводится к трем строкам кода, представляющим собой общую схему, повторяемую в большинстве пользовательских модальных диалогов:

`win.focus_set()`

Заставляет окно перехватить фокус ввода приложения, как если бы оно было сделано активным с помощью щелчка мыши. Для этого метода есть также синоним, `focus`, и часто фокус устанавливается не на все окно, а на графический элемент в нем, служащий для ввода данных (например, `Entry`).

`win.grab_set()`

Отключает все другие окна приложения, пока не будет уничтожено данное окно. Во время захвата пользователь не может взаимодействовать с другими окнами программы.

`win.wait_window()`

Приостанавливает вызвавшего, пока не будет уничтожен графический элемент `win`, но при этом главный цикл обработки событий (`mainloop`) остается активным. Это означает, что GUI в целом остается активным во время ожидания; например, его окна перерисовываются при скрытии под другими или открытии. Когда окно уничтожается с помощью метода `destroy`, оно удаляется с экрана, захват приложения автоматически освобождается и происходит возврат из этого метода.

Так как сценарий ждет события разрушения окна, он должен обеспечить уничтожение окна обработчиком обратного вызова в ответ на взаимодействие с графическими элементами в диалоговом окне (единственном, которое активно). Диалог в этом примере является просто информирующим, поэтому его кнопка OK вызывает метод `destroy` окна. В диалогах для ввода данных пользователем можно установить вместо этого обработчик обратного вызова для нажатия клавиши Enter, который возьмет данные, введенные в элемент `Entry`, и после этого вызовет `destroy` (как будет показано далее в этой главе).

Другие способы осуществления модальности

Модальные диалоги обычно реализуются путем создания нового всплывающего окна и ожидания для него события `destroy`, как в этом примере. Но существуют и другие схемы. Например, можно создать диалоговые окна заранее и по мере необходимости показывать или скрывать их с помощью методов окна верхнего уровня `deiconify` и `withdraw` (подробности смотрите в сценариях раздела «Средства синхронизации, потоки и анимация» главы 8). С учетом того, что в настоящее время скорость создания окон такова, что оно происходит практически мгновенно, такой способ встречается значительно реже, чем создание окон с нуля и разрушение их при каждом взаимодействии.

Можно также реализовать состояние модальности путем ожидания изменения значения *переменной* Tkinter, а не уничтожения окна. Подробности смотрите в последующем обсуждении переменных Tkinter в данной главе (они являются объектами классов, а не обычными переменными Python) и метода `wait_variable` в конце главы 8. В этой схеме обработчик обратного вызова диалогового окна с большим сроком жизни может подать сигнал об изменении состояния ожидающей головной программе без необходимости уничтожения диалогового окна.

Наконец, если вызвать метод `mainloop` рекурсивно, возврат из вызова произойдет только после выполнения метода `quit` графического элемента. Метод `quit` прекращает выполнение вызова `mainloop` и потому обычно завершает выполнение программы GUI. Но если есть действующий уровень рекурсии `mainloop`, он просто завершит его. Благодаря этому при желании модальные диалоги можно писать без вызова метода ожидания. Скажем, сценарий примера 7.14 работает так же, как `dlg-custom`.

Пример 7.14. PP2E\Gui\Tour\dlg-recursive.py

```
from Tkinter import *

def dialog():
    win = Toplevel()                                    # создать новое окно
    Label(win,  text='Hard drive reformatted!').pack()  # добавить несколько
                                                        # графических элементов
    Button(win, text='OK', command=win.quit).pack()     # установить обратный вызов quit
    win.protocol('WM_DELETE_WINDOW', win.quit)          # quit также при закрытии окна!

    win.focus_set()          # перехватить фокус ввода,
    win.grab_set()           # отключить другие окна,
    win.mainloop()           # запустить вложенный цикл событий для ожидания
    win.destroy()
    print 'dialog exit'

root = Tk()
Button(root, text='popup', command=dialog).pack()
root.mainloop()
```

Выбирая этот путь, нужно вместо `destroy` вызывать в обработчиках обратного вызова `quit` (`destroy` не завершает уровень `mainloop`) и обеспечить вызов `quit` кнопкой закры-

тия окна с помощью protocol (иначе не будет завершаться вызов рекурсивного уровня mainloop, что приведет к генерации странных сообщений об ошибках при окончательном выходе из программы). Из-за этой дополнительной сложности более удобным может оказаться использование wait_window или wait_variable, а не рекурсивных вызовов mainloop.

Как строить диалоговые окна типа форм с метками и полями ввода, мы увидим позже в этой главе, познакомившись с элементом Entry, и еще раз – при изучении менеджера grid в главе 8. Другие примеры пользовательских диалогов можно найти в ShellGui (глава 9), PyMailGui (глава 11), PyCalc (глава 18) и немодальном *form.py* (глава 10). А сейчас мы перейдем к более глубокому изучению событий, что окажется ценным на более поздних этапах обзора.

Привязка событий

В предыдущей главе мы познакомились с методом графических элементов bind, который использовался для перехвата нажатий кнопок. Так как bind часто используется вместе с другими графическими элементами (например, для перехвата нажатия клавиши Enter в окнах для ввода), остановимся на нем здесь в начале нашего обзора. Пример 7.15 иллюстрирует другие протоколы событий для bind.

Пример 7.15. PP2E\Gui\Tour\bind.py

```
from Tkinter import *

def showPosEvent(event):
    print 'Widget=%s X=%s Y=%s' % (event.widget, event.x, event.y)

def showAllEvent(event):
    print event
    for attr in dir(event):
        print attr, '=>', getattr(event, attr)

def onKeyPress(event):
    print 'Got key press:', event.char

def onArrowKey(event):
    print 'Got up arrow key press'

def onReturnKey(event):
    print 'Got return key press'

def onLeftClick(event):
    print 'Got left mouse button click:',
    showPosEvent(event)

def onRightClick(event):
    print 'Got right mouse button click:',
    showPosEvent(event)

def onMiddleClick(event):
    print 'Got middle mouse button click:',
    showPosEvent(event)
    showAllEvent(event)

def onLeftDrag(event):
    print 'Got left mouse button drag:',
    showPosEvent(event)

def onDoubleLeftClick(event):
```

```
    print 'Got double left mouse click',
    showPosEvent(event)
    tkroot.quit()

tkroot = Tk()
labelfont = ('courier', 20, 'bold')                # семейство, размер, стиль
widget = Label(tkroot, text='Hello bind world')
widget.config(bg='red', font=labelfont)            # красный фон, крупный шрифт
widget.config(height=5, width=20)                  # первоначальный размер строк, символов
widget.pack(expand=YES, fill=BOTH)

widget.bind('<Button-1>',  onLeftClick)            # щелчки кнопок мыши
widget.bind('<Button-3>',  onRightClick)
widget.bind('<Button-2>',  onMiddleClick)          # средняя - это обе в некоторых мышах
widget.bind('<Double-1>',  onDoubleLeftClick)      # двойной щелчок левой
widget.bind('<B1-Motion>', onLeftDrag)             # щелчок левой и перемещение

widget.bind('<KeyPress>',  onKeyPress)             # все нажатия клавиш
widget.bind('<Up>',        onArrowKey)             # нажата клавиша со стрелкой
widget.bind('<Return>',    onReturnKey)            # нажата клавиша return/enter
widget.focus()                                     # или привязать нажатие клавиши к tkroot
tkroot.title('Click Me')
tkroot.mainloop()
```

Этот файл состоит в основном из функций обработчиков обратного вызова, вызываемых при возникновении событий. Как было показано в главе 6, все эти обратные вызовы получают в качестве аргумента объект события, предоставляющий сведения о сгенерированном событии. Технически этот аргумент является экземпляром класса Tkinter `Event`, и содержащиеся в нем подробности представлены атрибутами; большинство обратных вызовов просто трассируют события, выводя соответствующие их атрибуты.

При запуске этого сценария создается окно, показанное на рис. 7.18; главное его назначение – служить областью для запуска событий щелчков мышью и нажатия клавиш.

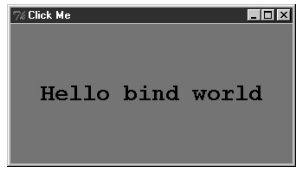


Рис. 7.18. Окно bind для щелчков мышью

Черно-белое издание, которое вы держите в руках, не слишком справедливо к этому сценарию – при живой работе он использует приведенные выше настройки и выводит текст черным по красному большим шрифтом Courier. Вам придется поверить мне на слово (или запустить его самим).

Но главная задача этого примера – продемонстрировать действие других протоколов связывания событий. Мы уже видели раньше сценарий, в котором с помощью метода `bind` графического элемента перехватывались одиночные и двойные щелчки левой кнопкой мыши; этот сценарий демонстрирует другие виды событий, часто перехватываемые `bind`:

`<KeyPress>`

Чтобы перехватывать нажатия одиночных клавиш на клавиатуре, зарегистрируйте обработчик для события с идентификатором `<KeyPress>`; это средство более низ-

кого уровня для ввода данных в программах GUI, чем графический элемент `Entry`, о котором рассказывается в следующем разделе. Нажатая клавиша возвращается в виде ASCII в объекте события, передаваемом обработчику обратного вызова (`event.char`). Другие атрибуты структуры события идентифицируют нажатую клавишу деталями более низкого уровня. Нажатия клавиш можно перехватывать графическим элементом – корневым окном верхнего уровня или графическим элементом, которому назначен фокус с помощью метода `focus`, используемого в данном сценарии.

`<B1-Motion>`

Этот сценарий перехватывает также перемещение мыши при нажатой кнопке: зарегистрированный обработчик события `<B1-Motion>` вызывается всякий раз, когда мышь передвигается при нажатой левой кнопке и получает в аргументе события (`event.x`, `event.y`) текущие координаты X/Y указателя мыши. Эти данные можно использовать для перемещения объекта, перетаскивания, рисования на уровне пикселов и т. д. (смотрите, например, PyDraw в главе 9).

`<Button-3>`, `<Button-2>`

Этот сценарий перехватывает также щелчки правой и средней кнопками мыши (называемыми также кнопками 3 и 2). Чтобы щелчок средней кнопкой 2 действовал для мыши с двумя кнопками, щелкните одновременно обеими кнопками; если это не действует, проверьте настройки мыши в интерфейсе свойств (Control Panel в Windows).[1]

`<Return>`, `<Up>`

Чтобы перехватывать более специфические нажатия клавиш, в данном сценарии зарегистрированы обработчики для событий нажатия клавиш Return/Enter и «стрелки вверх»; в противном случае эти события были бы отправлены общему обработчику `<KeyPress>` и потребовали анализа события.

Вот что попадает в выходной поток `stdout` после щелчка левой кнопкой, щелчка правой кнопкой, щелчка левой кнопкой и перетаскивания, нескольких нажатий клавиш, нажатия Return и «стрелки вверх» и, наконец, двойного щелчка левой кнопкой для завершения. При нажатии левой кнопки мыши и перемещении курсора по экрану возникает множество сообщений события перетаскивания – одно сообщение выводится для каждого движения при перетаскивании (и для каждого выполняется один обратный вызов Python):

```
C:\...\PP2E\Gui\Tour>python bind.py
Got left mouse button click: Widget=.7871632 X=209 Y=79
Got right mouse button click: Widget=.7871632 X=209 Y=79
Got left mouse button click: Widget=.7871632 X=83 Y=63
Got left mouse button drag: Widget=.7871632 X=83 Y=65
Got left mouse button drag: Widget=.7871632 X=84 Y=66
Got left mouse button drag: Widget=.7871632 X=85 Y=66
Got left mouse button drag: Widget=.7871632 X=85 Y=67
Got left mouse button drag: Widget=.7871632 X=85 Y=68
Got key press: s
Got key press: p
Got key press: a
Got key press: m
Got key press: 1
Got key press: -
```

[1] Видимо, у автора установлены какие-то дополнительные драйверы от производителя мыши. В стандартной поставке Windows такая настройка отсутствует. – *Примеч. ред.*

```
Got key press: 2
Got key press: .
Got return key press
Got up arrow key press
Got left mouse button click: Widget=.7871632 X=85 Y=68
Got double left mouse click Widget=.7871632 X=85 Y=68
```

Для событий, связанных с мышью, обратный вызов выводит координаты X и Y указателя мыши в передаваемом объекте события. Обычно координаты измеряются в пикселах от верхнего левого угла, но относительно того графического элемента, по которому произведен щелчок. Вот что выводится для щелчка левой кнопкой, средней и двойного щелчка левой. Обратите внимание на вывод целиком аргумента обработчиком обратного вызова для средней кнопки – всех атрибутов объекта `Event`. Различные типы событий устанавливают различные атрибуты; например большинство нажатий на клавиши помещают что-либо в `char`:

```
C:\...\PP2E\Gui\Tour>python bind.py
Got left mouse button click: Widget=.7871632 X=163 Y=18
Got middle mouse button click: Widget=.7871632 X=152 Y=110
<Tkinter.Event instance at 7b3640>
char => ??
height => 0
keycode => 2
keysym => ??
keysym_num => 2
num => 2
send_event => 0
serial => 14
state => 0
time => 5726238
type => 4
widget => .7871632
width => 0
x => 152
x_root => 156
y => 110
y_root => 133
Got left mouse button click: Widget=.7871632 X=152 Y=110
Got double left mouse click Widget=.7871632 X=152 Y=110
```

Помимо тех событий, которые проиллюстрированы в данном примере, есть другие виды связываемых событий, перехват которых может регистрироваться в сценарии Tkinter. Например:

- `<ButtonRelease>` генерируется при отпускании кнопки (`<ButtonPress>` выполняется, когда кнопка сначала нажимается).
- `<Motion>` срабатывает, когда перемещается указатель мыши.
- Обработчики `<Enter>` и `<Leave>` перехватывают вход и выход мыши из области отображения окна (полезно для автоматического выделения графического элемента).
- `<Configure>` вызывается при изменении размеров окна, его положения и т. д. (например, новые размеры окна задаются в атрибутах `width` и `height` объекта).
- `<Destroy>` вызывается при уничтожении графического элемента окна (и отличается от механизма `protocol` для нажатия кнопки закрытия менеджера окон).
- `<FocusIn>` и `<FocusOut>` выполняются, когда графический элемент получает или теряет фокус.

- <Map> и <Unmap> выполняются, когда окно открывается или превращается в значок.
- <Escape>, <BackSpace> и <Tab> перехватывают нажатия других специальных клавиш.
- <Down>, <Left> и <Right> перехватывают нажатия других клавиш со стрелками.

Этот список не полон, а для записи названий событий есть свой довольно сложный синтаксис, например:

- *Модификаторы* – могут добавляться к идентификаторам событий, чтобы сделать их еще более специфическими; например <B1-Motion> означает перемещение мыши при нажатой левой кнопке, а <KeyPress-a> указывает на нажатие только клавиши «a».
- *Синонимы* – могут использоваться для имен некоторых частых событий; например <ButtonPress-1>, <Button-1> и <1> означают нажатие левой кнопки мыши, а <KeyPress-a> и <Key-a> означают клавишу «a». Все форматы различают регистр: пишите <Key-Escape>, а не <KEY-ESCAPE>.
- Можно определять идентификаторы *виртуальных* событий, обозначающие последовательности из одного или нескольких событий, с помощью пары угловых скобок (например, <<PasteText>>).

С целью экономии места за исчерпывающими сведениями по этой теме мы отсылаем вас к другим справочным материалам по Tk и Tkinter. Кроме того, изменив настройки в сценарии и запустив его заново, можно также выяснить некоторые особенности поведения событий; в конце концов, это Python.

Message и Entry

Графические элементы Message и Entry позволяют отображать и вводить простой текст. Оба они, в сущности, являются функциональными подмножествами графического элемента Text, с которым мы познакомимся позднее, – Text может делать все то, что могут Message и Entry, но обратное неверно.

Message

Графический элемент Message служит просто местом для вывода текста. Хотя с помощью стандартного диалога showinfo, с которым мы встречались ранее, выводить всплывающие сообщения, вероятно, удобнее, Message автоматически и гибко разбивает длинные строки и может встраиваться внутрь элементов-контейнеров, когда нужно вывести на экране какой-либо текст только для чтения. Кроме того, этот графический элемент обладает более чем десятком параметров настройки, позволяющих изменять его внешний вид. Пример 7.16 и рис. 7.19 иллюстрируют основы применения Message; относительно других поддерживаемых параметров смотрите справочник по Tk или Tkinter.

Рис. 7.19. Графический элемент Message в действии

Пример 7.16. PP2E\Gui\tour\message.py

```
from Tkinter import *
msg = Message(text="Oh by the way, which one's Pink?")
msg.config(bg='pink', font=('times', 16, 'italic'))
msg.pack()
mainloop()
```

Entry

Графический элемент `Entry` служит простым полем для ввода одной строки текста. Обычно он используется в качестве полей ввода в диалоговых окнах типа форм и всюду, где пользователь должен ввести значение в поле более крупного изображения. `Entry` также поддерживает более сложные понятия, например прокрутку, привязку клавиш для редактирования и выделение текста, но на практике прост в использовании. Пример 7.17 создает окно для ввода, показанное на рис. 7.20.

Рис. 7.20. Entry1, схваченный с поличным в процессе ввода

Пример 7.17. PP2E\Gui\tour\entry1.py

```
from Tkinter import *
from quitter import Quitter

def fetch():
    print 'Input => "%s"' % ent.get()              # получить текст

root = Tk()
ent = Entry(root)
ent.insert(0, 'Type words here')                   # задать текст
ent.pack(side=TOP, fill=X)                         # раздвинуть по горизонтали

ent.focus()                                        # избавить от щелчка
ent.bind('<Return>', (lambda event: fetch()))      # по клавише <Enter>
btn = Button(root, text='Fetch', command=fetch)    # и по кнопке
btn.pack(side=LEFT)
Quitter(root).pack(side=RIGHT)
root.mainloop()
```

При запуске сценарий `entry1` заполняет текстовое поле этого GUI текстом «Type words here» с помощью вызова метода `insert` графического элемента. Поскольку обе кнопки, Fetch и Enter, запускают в сценарии функцию обратного вызова `fetch`, оба события пользователя получают и выводят текущий текст поля ввода с помощью метода `get`:

```
C:\...\PP2E\Gui\Tour>python entry1.py
Input => "Type words here"
Input => "Have a cigar"
```

Мы уже встречались ранее с событием `<Return>` при изучении `bind`; в отличие от нажатий на кнопки, эти обратные вызовы более низкого уровня получают в качестве аргумента событие, поэтому для его игнорирования в сценарии использована оборачивающая лямбда. В этом сценарии также поле ввода упаковывается с параметром `fill=X`,

чтобы оно растягивалось по горизонтали вместе с окном (попробуйте сами), и вызывается метод focus графического элемента, чтобы поле ввода получало фокус при появлении окна. Благодаря такой установке фокуса в коде пользователю не нужно щелкать по полю, чтобы начать ввод данных.

Программирование графических элементов Entry

Вообще говоря, значения, вводимые в элементы Entry и отображаемые ими, могут быть установлены или получены с помощью связанных объектов «переменных» (описываемых далее в этой главе) или с помощью следующих вызовов методов графического элемента Entry:

```
ent.insert(0, 'some text')          # установка значения
value = ent.get()                   # получение значения (строки)
```

Первый параметр метода insert задает позицию, в которую должен быть введен текст. Здесь «0» означает ввод впереди, поскольку смещения начинаются с нуля, а целое 0 и строка '0' означают одно и то же (аргументы методов Tkinter всегда при необходимости преобразуются в строки). Если элемент Entry уже содержит текст, то обычно требуется удалить его содержимое перед установкой нового значения, иначе новый текст будет просто добавлен к уже существующему:

```
ent.delete(0, END)                  # сперва удалить текст с начала до конца
ent.insert(0, 'some text')          # затем установить значение
```

Здесь имя END является предопределенной константой Tkinter, обозначающей конец графического элемента; она снова встретится нам в главе 8 при изучении полномасштабного и многострочного графического элемента Text (более мощного собрата Entry). Поскольку после удаления элемент пуст, предыдущая последовательность команд эквивалентна такой:

```
ent.delete('0', END)                # удалить текст с начала до конца
ent.insert(END, 'some text')        # добавить в конец пустого текста
```

В любом случае, если сначала не удалить текст, новый вводимый текст просто добавляется. Если хотите увидеть, как это происходит, измените функцию fetch так, как показано ниже, и при каждом нажатии кнопки или клавиши в начало и конец поля ввода будет добавляться «x»:

```
def fetch():
    print 'Input => "%s"' % ent.get()       # получить текст
    ent.insert(END, 'x')                    # для очистки: ent.delete('0', END)
    ent.insert(0, 'x')                      # новый текст просто добавляется
```

В последующих примерах мы встретимся также с параметром элемента Entry state='disabled', делающим его доступным только для чтения, а также параметром show='*', заставляющим его выводить каждый символ как * (полезно для ввода паролей). Испытайте это сами, изменив сценарий и запустив его. Entry поддерживает и другие параметры, которые мы здесь также пропустим; дополнительные сведения ищите в последующих примерах и других источниках.

Задание расположения элементов на формах для ввода

Как уже отмечалось, графические элементы Entry часто применяются при получении значений полей в окнах типа форм. Мы часто будем создавать такие формы в этой книге; простую иллюстрацию такого применения дает пример 7.18, в котором несколько меток и полей ввода объединены в форму для ввода нескольких значений, показанную на рис. 7.21.

Рис. 7.21. Вывод форм entry2 (и entry3)

Пример 7.18. PP2E\Gui\Tour\entry2.py

```
# Непосредственное использование элементов Entry и расположение по рядам

from Tkinter import *
from quitter import Quitter
fields = 'Name', 'Job', 'Pay'

def fetch(entries):
    for entry in entries:
        print 'Input => "%s"' % entry.get()         # получить текст

def makeform(root, fields):
    entries = []
    for field in fields:
        row = Frame(root)                           # создать новый ряд
        lab = Label(row, width=5, text=field)       # добавить метку, поле ввода
        ent = Entry(row)
        row.pack(side=TOP, fill=X)                  # упаковать ряд наверху
        lab.pack(side=LEFT)
        ent.pack(side=RIGHT, expand=YES, fill=X)    # расширить по горизонтали
        entries.append(ent)
    return entries

if __name__ == '__main__':
    root = Tk()
    ents = makeform(root, fields)
    root.bind('<Return>', (lambda event, e=ents: fetch(e)))
    Button(root, text='Fetch',
                 command=(lambda e=ents: fetch(e))).pack(side=LEFT)
    Quitter(root).pack(side=RIGHT)
    root.mainloop()
```

Полями ввода здесь служат простые элементы `Entry`. Сценарий строит явный список этих графических элементов, значения которых будут потом извлекаться. При каждом нажатии кнопки Fetch текущие значения во всех полях ввода захватываются и выводятся в стандартный выходной поток:

```
C:\...\PP2E\Gui\Tour>python entry2.py
Input => "Bob"
Input => "Technical Writer"
Input => "Jack"
```

Тот же вывод содержимого полей получается при нажатии клавиши Enter, когда это окно получает на экране фокус – на этот раз событие привязано к корневому окну в целом, а не к отдельному полю ввода.

Искусство создания структуры формы состоит в основном в организации иерархии графических элементов. В данном сценарии каждый ряд метка/поле ввода строится как новый `Frame`, прикрепляемый к текущему краю `TOP` окна; метки прикрепляются к

левому краю ряда (LEFT), а поля – к правому (RIGHT). Поскольку каждый ряд представляет собой отдельный Frame, его содержимое отделено от другой упаковки, производимой в этом окне. Этот сценарий также организует увеличение горизонтального размера при изменении размеров окна *только* для полей ввода, как на рис. 7.22.

Рис. 7.22. Расширение entry2 (и entry3) в действии

Снова создание модальных окон

Позднее мы увидим, как создавать аналогичные структуры форм с помощью менеджера геометрии grid. Но сейчас, взявшись за создание структуры форм, посмотрим, как применять технологию создания *модальных диалогов* к более сложным формам для ввода, таким как эта.

В примере 7.19 с помощью прежних функций makeform и fetch создается форма и выводится ее содержимое подобно тому, как это делалось раньше. Но теперь поля ввода прикрепляются к новому всплывающему окну Toplevel, создаваемому по требованию и содержащему кнопку OK, генерирующую событие разрушения окна. Как мы уже знаем, вызов wait_window влечет остановку до того времени, когда произойдет уничтожение.

Пример 7.19. PP2E\Gui\Tour\entry2-modal.py

```
# получить значения прежде, чем разрушить окно с полями ввода

from Tkinter import *
from entry2 import makeform, fetch, fields

def show(entries):
    fetch(entries)                  # нужно получить значения прежде, чем окно будет разрушено!
    popup.destroy()                 # данные пропадут, если порядок команд обратный

def ask():
    global popup
    popup = Toplevel()               # показать форму в окне модального диалога
    ents = makeform(popup, fields)
    Button(popup, text='OK', command=(lambda e=ents: show(e)) ).pack()
    popup.grab_set()
    popup.focus_set()
    popup.wait_window()              # ждать разрушения окна

root = Tk()
Button(root, text='Dialog', command=ask).pack()
root.mainloop()
```

При выполнении этого сценария нажатие кнопки в главном окне программы создает блокирующее окно диалога для заполнения формы, показанное на рис. 7.23.

Но в коде этого модального диалога таится тонкая опасность: поскольку он получает данные, вводимые пользователем, из графических элементов Entry, встроенных во всплывающее окно, нужно получить данные прежде, чем окно будет разрушено в обработчике обратного вызова для нажатия OK. Оказывается, что вызов destroy дейст-

Рис. 7.23. Окна, создаваемые entry2-modal (и entry3-modal)

вительно уничтожает все графические элементы разрушаемого окна; попытка получить значение из уничтоженного `Entry` не только не действует, но и порождает массу сообщений об ошибке в окне консоли – попробуйте изменить порядок команд в функции `show` и вы убедитесь в этом сами.

Чтобы избежать этой проблемы, нужно следить за тем, чтобы выборка значений осуществлялась перед разрушением, или использовать переменные Tkinter, являющиеся предметом следующего раздела.

«Переменные» Tkinter

Графические элементы `Entry` (наряду с другими) поддерживают понятие ассоциированной переменной; изменение ассоциированной переменной изменяет текст, показываемый в `Entry`, а изменение текста в `Entry` изменяет значение переменной. Однако это не обычные переменные Python – переменные, связанные с графическими элементами, являются экземплярами *классов* переменных в библиотеке модуля Tkinter. Эти классы носят названия `StringVar`, `IntVar`, `DoubleVar` и `BooleanVar`; выбор класса зависит от контекста, в котором он должен использоваться. Например, экземпляр класса `StringVar` может быть ассоциирован с полем `Entry`, как показано в примере 7.20.

Пример 7.20. PP2E\Gui\Tour\entry3.py

```
# Использование переменных StringVar и расположения по колонкам

from Tkinter import *
from quitter import Quitter
fields = 'Name', 'Job', 'Pay'

def fetch(variables):
    for variable in variables:
        print 'Input => "%s"' % variable.get()      # получить из переменных

def makeform(root, fields):
    form = Frame(root)                              # создать внешний фрейм
    left = Frame(form)                              # создать две колонки
    rite = Frame(form)
    form.pack(fill=X)
    left.pack(side=LEFT)
    rite.pack(side=RIGHT, expand=YES, fill=X)       # расширяться по горизонтали

    variables = []
    for field in fields:
        lab = Label(left, width=5, text=field)      # добавить в колонки
        ent = Entry(rite)
        lab.pack(side=TOP)
        ent.pack(side=TOP, fill=X)                  # расширяться по горизонтали
        var = StringVar()
        ent.config(textvariable=var)                # связать поле с переменной
        var.set('enter here')
```

```
            variables.append(var)
        return variables

    if __name__ == '__main__':
        root = Tk()
        vars = makeform(root, fields)
        Button(root, text='Fetch',
                     command=(lambda v=vars: fetch(v))).pack(side=LEFT)
        Quitter(root).pack(side=RIGHT)
        root.bind('<Return>', (lambda event, v=vars: fetch(v)))
        root.mainloop()
```

За исключением того обстоятельства, что поля ввода инициализируются строкой «enter here», этот сценарий создает окно, идентичное по внешнему виду и функциям тому, которое создает сценарий `entry2` (см. рис. 7.23). Для наглядности структура окна создается другим способом – как `Frame` с двумя вложенными фреймами, образующими левую и правую колонки в области формы, – но итоговый результат при выводе на экран оказывается тем же самым.

Главное, на что здесь нужно обратить внимание, это использование переменных `StringVar`. Вместо списка элементов `Entry`, из которого выбираются введенные значения, в этой версии хранится список объектов `StringVar`, которые ассоциируются с графическими элементами `Entry` следующим способом:

```
ent = Entry(rite)
var = StringVar()
ent.config(textvariable=var)                # связать поле с переменной
```

После того как переменные связаны, изменение и получение значения переменной:

```
var.set('text here')
value = var.get()
```

действительно изменяет и получает значение соответствующего поля ввода на экране.[1] Метод `get` переменного объекта возвращает строку для `StringVar`, целое число для `IntVar` и число с плавающей точкой для `DoubleVar`.

Конечно, как мы уже видели, можно легко устанавливать и получать текст непосредственно в полях `Entry`, без всякого дополнительного кода, использующего переменные. Зачем же нужно утруждать себя объектами переменных? Во-первых, исчезает опасность попытки получения значений после уничтожения, о чем говорилось в предыдущем разделе. Поскольку объекты `StringVar` продолжают жить после уничтожения графических элементов `Entry`, к которым они привязаны, можно извлекать из них значения тогда, когда модального диалога уже давно нет, как показано в примере 7.21.

Пример 7.21. PP2E\Gui\Tour\entry3-modal.py

```
# может получать значения после уничтожения окна с помощью stringvar

from Tkinter import *
from entry3 import makeform, fetch, fields

def show(variables):
```

[1] В устаревшем в данное время выпуске Tkinter, поставлявшемся с Python 1.3, можно было также устанавливать и получать переменные, вызывая их как функции, с аргументами или без (например, `var(value)` и `var()`). В настоящее время вместо этого нужно вызывать методы переменных `set` и `get`. По неустановленным причинам функциональная форма вызова прекратила работать годы назад, но ее еще можно встретить в старом коде Python (и первых изданиях по крайней мере одной книги O'Reilly по Python).

```
        popup.destroy()                 # порядок здесь не имеет значения
        fetch(variables)                # переменные сохраняются после уничтожения окна

    def ask():
        global popup
        popup = Toplevel()              # показ формы в модальном окне диалога
        vars = makeform(popup, fields)
        Button(popup, text='OK', command=(lambda v=vars: show(v)) ).pack()
        popup.grab_set()
        popup.focus_set()
        popup.wait_window()             # ждать уничтожения окна

    root = Tk()
    Button(root, text='Dialog', command=ask).pack()
    root.mainloop()
```

Эта версия такая же, как исходная (показанная в примере 7.19 и на рис. 7.25), но теперь `show` уничтожает всплывающее окно *перед* тем, как введенные данные получены из `StringVar`s в списке, созданном `makeform`. Иными словами, переменные несколько более надежны в некоторых контекстах, потому что они не являются частью действительного дерева отображения. Например, они также ассоциируются с флажками, группами переключателей и ползунками, обеспечивая доступ к текущим значениям и связывая вместе несколько графических элементов. Случайно или нет, им посвящен следующий раздел.

Флажки, переключатели и ползунки

Этот раздел знакомит с тремя типами графических элементов – `Checkbutton` («флажок», графический элемент для задания нескольких выбранных вариантов), `Radiobutton` («переключатель», элемент для выбора одного варианта) и `Scale` («масштаб», иногда называемый «slider» – «ползунок»). Все они являются вариациями на одну тему и в какой-то мере связаны с простыми кнопками, поэтому мы будем изучать их здесь вместе. Чтобы тренироваться с этими элементами было интереснее, мы повторно используем модуль `dialogTable`, показанный в примере 7.8, для создания обратных вызовов при выборе графических элементов (обратные вызовы создают всплывающие диалоговые окна). Попутно мы воспользуемся только что рассмотренными переменными Tkinter для связи со значениями состояний этих графических элементов.

Флажки

Графические элементы `Checkbutton` и `Radiobutton` разработаны как ассоциируемые с переменными Tkinter: нажатие на кнопку изменяет состояние переменной, а установка переменной изменяет состояние кнопки, к которой она привязана. В действительности переменные Tkinter лежат в основе действия этих графических элементов:

- Группа *флажков* реализует интерфейс с выбором нескольких вариантов путем присвоения каждой кнопке (флажку) собственной переменной.
- Группа *переключателей* накладывает модель взаимоисключающего одиночного выбора путем придания каждой кнопке уникального значения и назначения одной и той же переменной Tkinter.

У обоих типов кнопок есть параметры `command` и `variable`. Параметр `command` позволяет зарегистрировать обратный вызов, выполняемый сразу при возникновении события нажатия кнопки, подобно обычным графическим элементам `Button`. Но, ассоциируя переменную Tkinter с параметром `variable`, можно также в любой момент получать

или изменять состояние графического элемента путем получения или изменения значения связанной с ним переменной.

Несколько проще флажки Tkinter, поэтому с них и начнем. Пример 7.22 создает группу из пяти флажков, показанную на рис.7.24. Для большей пользы он также добавляет кнопку, с помощью которой выводится текущее состояние всех флажков, и прикрепляет экземпляр кнопки Quitter, которую мы построили в начале обзора.

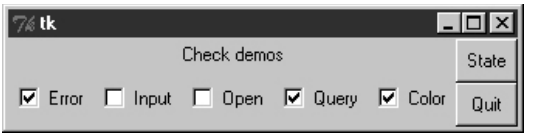


Рис. 7.24. demoCheck в действии

Пример 7.22. PP2E\Gui\Tour\demoCheck.py

```
from Tkinter import *              # получить базовый набор графических элементов
from dialogTable import demos      # получить готовые диалоги
from quitter import Quitter        # прикрепить к «себе» объект quitter

class Demo(Frame):
    def __init__(self, parent=None, **args):
        Frame.__init__(self, parent, args)
        self.pack()
        self.tools()
        Label(self, text="Check demos").pack()
        self.vars = []
        for key in demos.keys():
            var = IntVar()
            Checkbutton(self,
                        text=key,
                        variable=var,
                        command=demos[key]).pack(side=LEFT)
            self.vars.append(var)
    def report(self):
        for var in self.vars:
            print var.get(),    # текущие значения переключателей: 1 или 0
        print
    def tools(self):
        frm = Frame(self)
        frm.pack(side=RIGHT)
        Button(frm, text='State', command=self.report).pack(fill=X)
        Quitter(frm).pack(fill=X)

if __name__ == '__main__': Demo().mainloop()
```

В отношении программного кода флажки похожи на обычные кнопки, они даже упаковываются в графический элемент-контейнер. Однако по своему функционированию они несколько отличаются. Как можно сказать, глядя на рисунок (а лучше – запустив пример), флажок работает как переключатель: нажатие на него изменяет его состояние из выключенного во включенное (невыбранное в выбранное) или обратно из включенного в выключенное. Когда флажок выбран, на нем выводится галочка, а связанная с ним переменная `IntVar` получает значение 1; когда он не выбран, окошко пусто, а его переменная `IntVar` равна 0.

Чтобы смоделировать содержащее флажки приложение, кнопка State на этом экране запускает в сценарии метод report, который выводит текущие состояния всех пяти флажков в поток stdout. Вот вывод, получаемый после нескольких нажатий:

```
C:\...\PP2E\Gui\Tour>python demoCheck.py
0 0 0 0 0
1 0 0 0 0
1 0 1 0 0
1 0 1 1 0
1 0 0 1 0
1 0 0 1 1
```

В действительности это значения пяти переменных Tkinter, ассоциированных с флажками посредством параметров variable, но при опросе они дают значения кнопок. В этом сценарии с каждым из флажков на экране ассоциирована переменная Intvar, поскольку это двоичные индикаторы 0 или 1. Переменные StringVar тоже будут здесь работать, хотя их методы возвратят строки «0» или «1», а не целые числа, а их начальным состоянием будет пустая строка, а не целое число 0.

Параметр command этого графического элемента позволяет зарегистрировать обратный вызов, который будет выполняться при каждом нажатии кнопок. Для иллюстрации в качестве обработчика для каждого из флажков в этом сценарии зарегистрирован вызов демонстрации стандартного диалога: нажатие на кнопку изменяет состояние переключателя, а кроме того, выводит одно из знакомых диалоговых окон.

Интересно, что выполнить метод report можно также интерактивно. При работе в таком режиме элементы показываются во всплывающем окне при вводе строк и полностью действуют даже без вызова mainloop:

```
C:\...\PP2E\Gui\Tour>python
>>> from demoCheck import Demo
>>> d = Demo()
>>> d.report()
0 0 0 0 0
>>> d.report()
1 0 0 0 0
>>> d.report()
1 0 0 1 1
```

Флажки и переменные

Когда я впервые изучал этот графический элемент, моей первой реакцией было: «Зачем вообще здесь нужны переменные Tkinter, если можно зарегистрировать обратные вызовы для нажатий на кнопки?» На первый взгляд связанные переменные могут показаться излишними, но они упрощают некоторые действия с GUI. Не буду просить принять это на веру, а постараюсь объяснить, почему.

Помните, что обратный вызов для флажка, заданный в command, будет выполняться при *каждом* нажатии – при переключении и в выбранное и в невыбранное состояние. Поэтому, если нужно совершить действие немедленно после нажатия флажка, как правило, в обработчике обратного вызова требуется узнать текущее значение кнопки. Поскольку у флажка нет метода «get» для получения значений, обычно требуется запрашивать ассоциированную переменную, чтобы узнать, включена кнопка или выключена.

Кроме того, некоторые GUI разрешают пользователям устанавливать флажки, вообще не выполняя при этом обратных вызовов, и получить значения кнопок где-либо позже в программе. В таком сценарии переменные служат для автоматического запо-

минания состояний кнопок. Представителем этого последнего подхода является метод `report` в сценарии `demoCheck`.

Конечно, можно и вручную запоминать состояние каждой кнопки в обработчиках обратных вызовов. В примере 7.23 ведется свой список состояний переключателей, который вручную обновляется в функциях обратных вызовов, заданных в `command`.

Пример 7.23. PP2E\Gui\Tour\demo-check-manual.py

```
# флажки, трудный способ (без переменных)

from Tkinter import *
states = []
def onPress(i):                         # запоминание состояния
    states[i] = not states[i]           # изменяет 0->1, 1->0

root = Tk()
for i in range(10):
    chk = Checkbutton(root, text=str(i), command=(lambda i=i: onPress(i)) )
    chk.pack(side=LEFT)
    states.append(0)
root.mainloop()
print states                            # при выходе показать все состояния
```

Здесь лямбда передает индекс нажатой кнопки в список `states` (иначе для каждой кнопки потребовалась бы отдельная функция обратного вызова). При запуске этот сценарий создает окно с 10-ю флажками, как на рис. 7.25.

Рис. 7.25. Окно флажков с ручным запоминанием состояний

Поддерживаемые вручную состояния флажков обновляются при каждом нажатии кнопки и выводятся при выходе из GUI (формально, при возврате из вызова `mainloop`):

```
C:\...\PP2E\Gui\Tour>python demo-check-manual.py
[0, 0, 1, 0, 1, 0, 0, 0, 1, 0]
```

Такой способ действует, и его не столь уж трудно реализовать. Но связанные переменные Tkinter заметно облегчают эту задачу, особенно если до какого-то момента в будущем нет необходимости обрабатывать состояния флажков. Это проиллюстрировано в примере 7.24.

Пример 7.24. PP2E\Gui\Tour\demo-check-auto.py

```
# флажки, простой способ обработки

from Tkinter import *
root = Tk()
states = []
for i in range(10):
    var = IntVar()
    chk = Checkbutton(root, text=str(i), variable=var)
    chk.pack(side=LEFT)
    states.append(var)
root.mainloop()                             # пусть работает Tkinter
print map((lambda var: var.get()), states)  # при выходе показать все состояния
```

Внешний вид и функционирование такие же, но command с указанием обработчика обратного вызова по нажатию кнопки вообще нет, потому что Tkinter автоматически отслеживает изменение состояний:

```
C:\...\PP2E\Gui\Tour>python demo-check-auto.py
[0, 0, 1, 0, 0, 0, 1, 0, 0, 0]
```

Смысл здесь в том, что необязательно связывать переменные с флажками, но если сделать это, то работать с GUI будет легче.

Переключатели

Переключатели обычно используются группами: так же, как для механических кнопок выбора станций в радиоприемниках прошлого, нажатие одного графического элемента Radiobutton из группы автоматически делает невыбранными все кнопки, кроме той, которая нажата последней. Иными словами, одновременно может быть выбрано не более одной кнопки. В Tkinter связывание всех переключателей в группе с уникальными значениями и одной и той же переменной гарантирует, что в каждый данный момент времени может быть выбрано не более одного из них.

Подобно кнопкам флажков и обычным кнопкам, переключатели поддерживают параметр command для регистрации обратного вызова, обрабатывающего нажатие немедленно. Подобно флажкам, у переключателей также есть атрибут variable для связывания кнопок в группу и получения текущего выбора в произвольный момент времени.

Кроме того, у переключателей есть атрибут value, позволяющий сообщить Tkinter, какое значение должна иметь ассоциированная с кнопкой переменная, когда выбирается эта кнопка. Поскольку с одной и той же переменной ассоциируется более одного переключателя, необходимо точно определить значение каждой кнопки (это не просто схема с переключением 1 или 0). В примере 7.25 демонстрируются основы использования переключателей.

Пример 7.25. PP2E\Gui\Tour\demoRadio.py

```
from Tkinter import *                # получить базовый набор графических элементов
from dialogTable import demos        # обработчики обратных вызовов кнопок
from quitter import Quitter          # прикрепить к «себе» объект quitter

class Demo(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        Label(self, text="Radio demos").pack(side=TOP)
        self.var = StringVar()
        for (key, value) in demos.items():
            Radiobutton(self, text=key,
                              command=self.onPress,
                              variable=self.var,
                              value=key).pack(anchor=NW)
        Button(self, text='State', command=self.report).pack(fill=X)
        Quitter(self).pack(fill=X)
    def onPress(self):
        pick = self.var.get()
        print 'you pressed', pick
        print 'result:', demos[pick]()
    def report(self):
        print self.var.get()

if __name__ == '__main__': Demo().mainloop()
```

На рис. 7.26 показано, что создается при выполнении этого сценария. Нажатие на любой из переключателей в этом окне запускает обработчик `command`, показывает один из стандартных диалогов, с которыми мы знакомы, и автоматически делает невыбранной кнопку, которая нажималась перед этим. Как и флажки, переключатели упаковываются; в данном сценарии они упаковываются к верхнему краю, располагаясь по вертикали, а затем выравниваются, прикрепляясь к северо-западному углу отведенного им пространства.

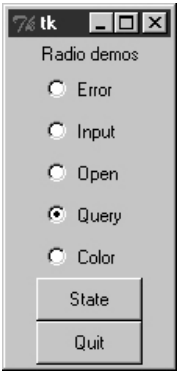


Рис. 7.26. demoRadio в действии

Как и в демонстрационном сценарии для флажков, кнопка State служит для запуска метода класса `report` и показа текущего состояния кнопок (выбранной кнопки). В отличие от демонстрационного сценария для флажков в этом сценарии выводятся также значения, возвращаемые вызовами диалогов, осуществляющимися при нажатии кнопок. Вот как выглядит поток `stdout` после нескольких нажатий кнопок и дампов состояний; состояния выделены полужирным шрифтом:

```
C:\...\PP2E\Gui\Tour>python demoRadio.py
you pressed Input
result: 3.14
Input
you pressed Open
result: C:/PP2ndEd/examples/PP2E/Gui/Tour/demoRadio.py
Open
you pressed Query
result: yes
Query
```

Переключатели и переменные

Так зачем здесь нужны переменные? С одной стороны, у переключателей тоже нет метода графических элементов «get» для получения значения выбора. Еще более важно то, что в группах переключателей установка `value` и `variable` лежит в основе всего режима выбора одного варианта. На самом деле для того, чтобы переключатели вообще могли работать, решающим оказывается то, что все они ассоциируются с *одной и той же* переменной Tkinter и все имеют *различные* значения. Однако, чтобы действительно разобраться в причинах, нужны еще некоторые сведения о том, как действуют переключатели и переменные.

Как уже было показано, при изменении графического элемента изменяется ассоциируемая с ним переменная Tkinter, и наоборот. Но верно и то, что любое изменение

переменной автоматически изменяет *каждый* графический элемент, с которым она связана. При работе с переключателями нажатие на один из них устанавливает значение совместно используемой переменной, которая, в свою очередь, оказывает воздействие на другие кнопки, ассоциированные с этой переменной. При условии, что у всех переключателей различные значения, это вызывает ожидаемый эффект: когда в результате нажатия на кнопку значение совместно используемой переменной изменяется на значение нажатой кнопки, все остальные кнопки оказываются невыбранными, потому что значение переменной оказывается несовпадающим с их значениями.

Такой эффект распространения является довольно тонким, но будет легче понять его, зная, что если в множестве переключателей, связанных с одной и той же переменной, назначить некоторой группе переключателей *одно и то же* значение, то вся эта группа окажется выбранной, если нажать на любую из них. Рассмотрим пример 7.26 и рис. 7.27: поскольку у всех переключателей – 0, 3, 6 и 9 – значение 0 (остаток от деления на 3), при выборе любого из них выбираются они все (рис. 7.27).

Рис. 7.27. Переключатели испортились?

Пример 7.26. PP2E\Gui\Tour\demo-radio-multi.py

```
# что происходит, когда у некоторых кнопок одинаковое значение
from Tkinter import *
root = Tk()
var = StringVar()
for i in range(10):
    rad = Radiobutton(root, text=str(i), variable=var, value=str(i % 3))
    rad.pack(side=LEFT)
root.mainloop()
```

Если теперь нажать кнопку 1, 4 или 7, будут выбраны все три, а предыдущие выбранные окажутся сброшенными (их значение не равно «1»). Обычно это не то, что требуется, поэтому следите за тем, чтобы всем кнопкам была назначена одна и та же переменная, но разные значения, если хотите, чтобы переключатели действовали, как им положено. Например, в сценарии `demoRadio` имя демонстрационной программы дает естественное уникальное значение для каждой кнопки.

Переключатели без переменных

В примере 7.27 также реализована модель с одним выбором, но без переменных, путем выбора и сброса элементов в группе вручную в обработчике обратного вызова. При каждом событии нажатия вызывается `deselect` для всех объектов в группе и `select` для того, который нажат.

Пример 7.27. PP2E\Gui\Tour\demo-radio-manual.py

```
# переключатели, реализованные трудным способом (без переменных). Учтите, что deselect
# для переключателя устанавливает связанное с кнопкой значение равным нулевой строке,
# поэтому нужно либо все равно дать кнопкам уникальные значения, либо использовать флажки;
from Tkinter import *
state = ''
buttons = []
def onPress(i):
    global state
```

```
    state = i
    for btn in buttons:
        btn.deselect()
    buttons[i].select()
root = Tk()
for i in range(10):
    rad = Radiobutton(root, text=str(i), value=str(i), command=(lambda i=i: onPress(i)))
    rad.pack(side=LEFT)
    buttons.append(rad)
root.mainloop()
print state               # вывести состояние при выходе
```

Этот сценарий создает такое же окно с 10-ю переключателями, как на рис. 7.27, но реализует интерфейс с однозначным выбором, причем текущее состояние хранится в глобальной переменной Python, которая выводится при выходе из сценария. Все это Tkinter может сделать вместо вас, если использовать связанную переменную Tkinter и уникальные значения, как в примере 7.28.

Пример 7.28. PP2E\Gui\Tour\demo-radio-auto.py

```
# простой способ работы с переключателями
from Tkinter import *
root = Tk()                     # IntVar тоже можно использовать
var  = IntVar()                 # state = var.get()
for i in range(10):
    rad = Radiobutton(root, text=str(i), value=i, variable=var)
    rad.pack(side=LEFT)
root.mainloop()
print var.get()                 # вывести состояние при выходе
```

Работает точно так же, но вводить и отлаживать значительно проще. Обратите внимание на связывание в этом сценарии кнопок с `IntVar`, целочисленным собратом `StringVar`; если значения кнопок уникальны, можно также пользоваться и целыми числами.

Берегите свои переменные

Небольшое предостережение: в целом следует сохранять объект переменной Tkinter, используемой для связи с переключателями в течение всего времени показа переключателей. Присвойте его глобальной переменной модуля, запомните в структуре данных с длительным временем существования или сохраните как атрибут долго живущего объекта класса, как сделано в `demoRadio`, – живущего, пока вы как-то сохраняете ссылку на него. Обычно вы каким-то образом получите состояние, и вряд ли вас когда-нибудь коснется то, о чем я хочу сказать.

В текущей версии Tkinter классы переменных обладают деструктором `__del__`, который автоматически сбрасывает созданную переменную `Tk`, когда уничтожается объект Python (то есть попадает в уборку мусора). В итоге все ваши переключатели могут оказаться невыбранными, если объект переменной попадет в мусор, по крайней мере, до того момента, когда очередное нажатие кнопки установит для переменной `Tk` новое значение. Пример 7.29 показывает ситуацию, в которой это может случиться.

Пример 7.29. PP2E\Gui\Tour\demo-radio-clear.py

```
# берегите переменные переключателей (действительно незаметная вещь)
from Tkinter import *
root = Tk()
def radio1():                   # локальные переменные являются временными
    #global tmp                 # проблема решается, если сделать tmp глобальной
```

```
        tmp = IntVar()
        for i in range(10):
            rad = Radiobutton(root, text=str(i), value=i, variable=tmp)
            rad.pack(side=LEFT)
        tmp.set(5)
    radio1()
    root.mainloop()
```

Кажется, что первоначально должна быть выбрана кнопка 5, но этого не происходит. Локальная переменная `tmp` уничтожается при выходе из функции, переменная `Tk` сбрасывается, и значение 5 теряется (все кнопки оказываются невыбранными). Тем не менее эти переключатели прекрасно работают, когда начинаешь их нажимать, поскольку при этом переустанавливается переменная `Tk`. После раскомментирования команды `global` кнопка 5 появляется в выбранном состоянии, как и хотелось.

Конечно, это нетипичный пример – в таком виде невозможно узнать, какая кнопка нажата, потому что переменная не сохраняется (и `command` не установлена). Это настолько невразумительно, что я отсылаю вас к примеру *demo-radio-clear2.py* на компакт-диске, в котором делается попытка другими способами заставить проявиться эту странность. Возможно, вам это не понадобится, но если вы столкнетесь с этим, не говорите, что я вас не предупредил.

Ползунки

Ползунки («scales» или «sliders») используются для выбора значения из диапазона чисел. Перемещение положения ползунка с помощью перетаскивания или щелчка мышью изменяет значение графического элемента в диапазоне целых чисел и запускает обратный вызов Python, если он зарегистрирован.

Подобно флажкам и переключателям, ползунки обладают параметром `command` для регистрации управляемого событиями обработчика обратного вызова, выполняемого немедленно при перемещении ползунка, а также параметром `variable` для установления связи с переменной Tkinter, которая позволяет в любой момент времени получить или установить положение ползунка. Обрабатывать значение можно тогда, когда оно установлено, или позже.

Кроме того, у ползунков есть третий вариант обработки – методы `get` и `set`, с помощью которых в сценарии можно непосредственно обращаться к значениям элемента, не связывая с ним переменную. Поскольку обратные вызовы, регистрируемые `command`, получают текущее значение ползунка в качестве аргумента, часто этого достаточно, чтобы не прибегать к связанным переменным или вызовам методов `get/set`.

Для иллюстрации основ применения этого элемента в примере 7.30 создаются два ползунка – горизонтальный и вертикальный, которые связываются между собой через ассоциированную переменную, что позволяет их синхронизировать.

Пример 7.30. PP2E\Gui\Tour\demoScale.py

```
from Tkinter import *                # получить базовый набор графических элементов
from dialogTable import demos        # обработчики обратного вызова кнопки
from quitter import Quitter          # прикрепить фрейм для завершения
class Demo(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        Label(self, text="Scale demos").pack()
        self.var = IntVar()
        Scale(self, label='Pick demo number',
```

```
                    command=self.onMove,                    # перехват перемещения
                    variable=self.var,                      # отражает положение
                    from_=0, to=len(demos)-1).pack()
        Scale(self, label='Pick demo number',
                    command=self.onMove,                    # перехват перемещения
                    variable=self.var,                      # отражает положение
                    from_=0, to=len(demos)-1,
                    length=200, tickinterval=1,
                    showvalue=YES, orient='horizontal').pack()
        Quitter(self).pack(side=RIGHT)
        Button(self, text="Run demo", command=self.onRun).pack(side=LEFT)
        Button(self, text="State",    command=self.report).pack(side=RIGHT)
    def onMove(self, value):
        print 'in onMove', value
    def onRun(self):
        pos = self.var.get()
        print 'You picked', pos
        pick = demos.keys()[pos]    # отображение позиции на ключ
        print demos[pick]()
    def report(self):
        print self.var.get()
if __name__ == '__main__':
    print demos.keys()
    Demo().mainloop()
```

Кроме доступа к значению и регистрации обратного вызова у ползунков есть параметры, соответствующие понятию диапазона выбираемых значений, большинство из которых продемонстрировано в коде этого примера:

- Параметр `label` задает текст, появляющийся рядом со шкалой, `length` задает начальный размер в пикселах, а `orient` указывает направление.
- Параметры `from_` и `to` задают минимальное и максимальное значения шкалы («from» является зарезервированным словом Python, а «from_» – нет).
- Параметр `tickinterval` задает количество единиц измерения между отметками, наносимыми рядом со шкалой через равные интервалы (0 по умолчанию означает, что отметки не делаются).
- Параметр `resolution` задает количество единиц, на которое изменяется значение ползунка при каждом перетаскивании или щелчке левой кнопки мыши (по умолчанию 1).
- Параметр `showvalue` определяет, показывается ли текущее значение шкалы рядом с ползунком (по умолчанию `showvalue=YES`, что означает – показывается).

Обратите внимание, что ползунки тоже упаковываются в контейнере, как и прочие графические элементы Tkinter. Посмотрим, как эти понятия реализуются на практике; рис. 7.28 показывает окно, создающееся при выполнении этого сценария в Windows (в Unix и Mac получается аналогичная картина).

Для наглядности кнопка State показывает текущие значения ползунков, а «Run demo» запускает те же стандартные диалоги, используя целочисленные значения ползунков в качестве индекса таблицы demos. Сценарий регистрирует также обработчик `command`, который вызывается при каждом перемещении по любой из шкал и выводит новые положения ползунков. Вот сообщения, посылаемые на `stdout` после нескольких перемещений, запуск демонстрационных диалогов (курсив) и запрос состояния (полужирный):

```
C:\...\PP2E\Gui\Tour>python demoScale.py
['Error', 'Input', 'Open', 'Query', 'Color']
```

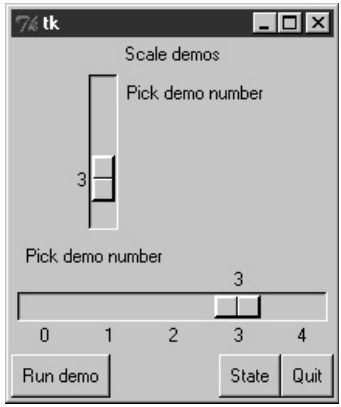


Рис. 7.28. demoScale в действии

```
in onMove 0
in onMove 0
in onMove 1
1
in onMove 2
You picked 2
C:/PP2ndEd/examples/PP2E/Gui/Tour/demoScale.py
in onMove 3
3
You picked 3
yes
```

Ползунки и переменные

Как можно догадаться, ползунки предлагают различные способы обработки их значений: сразу в обратном вызове при перемещении или позже путем получения текущего положения через переменные или вызовы методов. В действительности переменные Tkinter вообще не нужны для программирования ползунков – достаточно зарегистрировать обратные вызовы для перемещения или вызывать метод ползунка `get`, чтобы при необходимости получить значение шкалы, как в более простом примере 7.31.

Пример 7.31. PP2E\Gui\Tour\demo-scale-simple.py

```
from Tkinter import *
root = Tk()
scl = Scale(root, from_=-100, to=100, tickinterval=50, resolution=10)
scl.pack(expand=YES, fill=Y)
def report(): print scl.get()
Button(root, text='state', command=report).pack(side=RIGHT)
root.mainloop()
```

На рис. 7.29 показаны два экземпляра этой программы, выполняемые под Windows, один растянутый, а другой – нет (ползунки упакованы так, чтобы при изменении размера окна изменять размер по вертикали). Шкала выводит диапазон от –100 до 100, использует параметр `resolution` для изменения текущей позиции на 10 единиц вверх или вниз при каждом движении и устанавливает параметр `tickinterval` так, чтобы показывать рядом со шкалой значения с шагом 50. При нажатии в окне этого сценария кнопки State вызывается метод шкалы `get` для вывода текущего значения без всяких переменных или обратных вызовов:

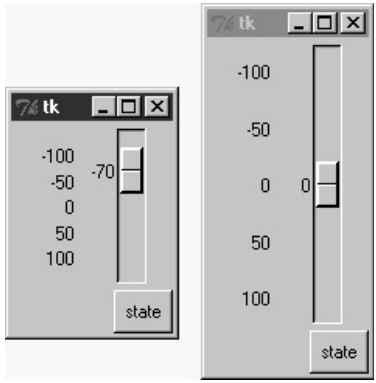


Рис. 7.29. Простая шкала без переменных

```
C:\...\PP2E\Gui\Tour>python demo-scale-simple.py
0
60
-70
```

Честно говоря, единственная причина использования переменных Tkinter в сценарии `demoScale` — это синхронизация шкал. Чтобы сделать пример более интересным, в этом сценарии один и тот же объект переменной Tkinter связан с обеими шкалами. Как было показано в предыдущем разделе, изменение графического элемента влечет за собой изменение его переменной, но изменение переменной изменяет *все* связанные с ней графические элементы. В отношении ползунков это означает, что перемещение ползунка обновляет переменную, которая, в свою очередь, может обновить все другие графические элементы, связанные с той же переменной. Так как в этом сценарии одна переменная связана с двумя шкалами, они автоматически становятся синхронизированными: перемещение одного ползунка вызывает перемещение другого, потому что при этом изменяется совместно используемая переменная, побочным эффектом чего является обновление другого ползунка.

Ваши приложения могут использовать или не использовать подобное связывание ползунков (граничащее с глубокой магией), но если задуматься, то это очень мощное средство. Связывая на экране несколько графических элементов с помощью переменных Tkinter, можно автоматически синхронизировать их, не трогая обработчики обратных вызовов. С другой стороны, синхронизацию можно обеспечить и без совместно используемой переменной, вызывая метод `set` одного ползунка из обработчика обратного вызова другого. Такое изменение вручную я оставляю читателю в качестве упражнения. То, что для одного глубокая магия, для другого может оказаться злостным хакерством.

Три способа выполнения кода GUI

После того как мы создали ряд похожих программ для запуска демонстраций, напишем несколько объединяющих их сценариев верхнего уровня. Поскольку демонстрационные программы были написаны так, что их можно применять в виде классов или сценариев, они могут быть использованы как компоненты, прикрепляемые к фреймам, запускаться в собственных окнах верхнего уровня или как самостоятельные программы. Все три варианта иллюстрируют принцип *многократного использования кода* в действии.

Прикрепление к фреймам

Чтобы проиллюстрировать иерархическое построение GUI в более крупном масштабе, чем это делалось до сих пор, пример 7.32 объединяет все четыре сценария панелей запуска диалогов из этой главы в одном фрейме. Он повторно использует код примеров 7.9, 7.22, 7.25 и 7.30.

Пример 7.32. PP2E\Gui\Tour\demoAll-frm.py

```
##############################################################################################
# 4 демонстрационных компонента классов (вложенные фреймы) в одном окне; в одном этом окне
# есть также 5 кнопок Quitter; gui могут повторно использоваться как фреймы, окна, процессы;
##############################################################################################
from Tkinter import *
from quitter import Quitter
demoModules = ['demoDlg', 'demoCheck', 'demoRadio', 'demoScale']
parts = []
def addComponents(root):
    for demo in demoModules:
        module = __import__(demo)                    # импорт по строке имени
        part = module.Demo(root)                     # прикрепить экземпляр
        part.config(bd=2, relief=GROOVE)
        part.pack(side=LEFT, fill=BOTH)
        parts.append(part)                           # изменить список тут же
def dumpState():
    for part in parts:                               # запустить отчет, если он есть
        print part.__module__ + ':',
        if hasattr(part, 'report'):
           part.report()
        else:
           print 'none'
root = Tk()                                          # окно toplevel по умолчанию
Label(root, text='Multiple Frame demo', bg='white').pack()
Button(root, text='States', command=dumpState).pack(fill=X)
Quitter(root).pack(expand=YES, fill=X)
addComponents(root)
mainloop()
```

Поскольку все четыре демонстрационные панели запуска запрограммированы так, что могут прикрепляться к контейнерам родительских графических элементов, это делается просто: нужно лишь передать один и тот же родительский графический элемент (в данном случае окно `root`) во все четыре вызова конструкторов демонстрационных программ и упаковать и настроить объекты демонстраций желаемым образом. Рис. 7.30 показывает, как выглядит результат – одно окно, в которое встроены экземпляры всех четырех знакомых нам демонстрационных программ запуска диалогов.

Конечно, этот пример является искусственным, но он демонстрирует мощь использования композиции при создании больших экранов GUI. Если представить себе, что каждый из прикрепленных демонстрационных объектов может быть чем-нибудь более полезным, например текстовым редактором, калькулятором или часами, то легче оценить значение этого примера.

Кроме фреймов демонстрационных объектов это составное окно содержит не менее пяти экземпляров написанной ранее кнопки Quitter (любая из них может закрыть этот GUI) и кнопку States для вывода текущих значений сразу всех встроенных демонстрационных объектов (она вызывает метод `report` для каждого объекта, у которого он есть). Вот пример того, какой вывод появится в потоке `stdout` после взаимодействия с графическими элементами этого экрана; вывод States показан полужирным шрифтом:

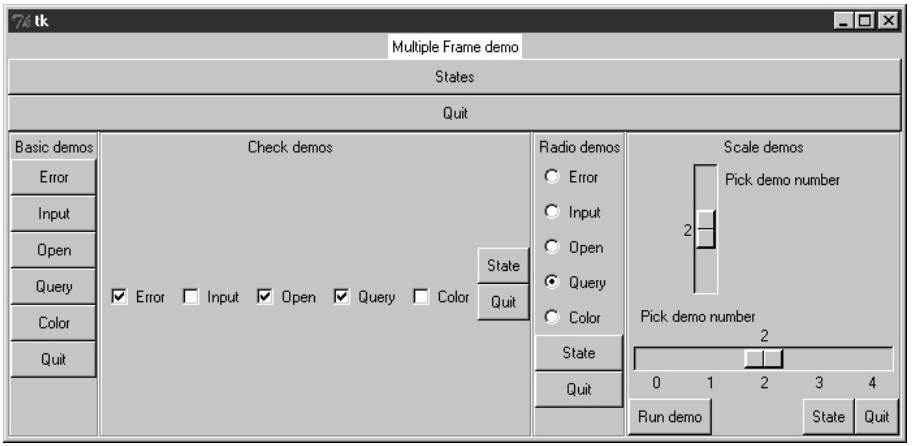


Рис. 7.30. demoAll_frm: вложенные фреймы

```
C:\...\PP2E\Gui\Tour>python demoAll_frm.py
in onMove 0
in onMove 0
demoDlg: none
demoCheck: 0 0 0 0 0
demoRadio:
demoScale: 0
you pressed Input
result: 1.234
demoDlg: none
demoCheck: 1 0 1 1 0
demoRadio: Input
demoScale: 0
you pressed Query
result: yes
in onMove 1
in onMove 2
You picked 2
C:/PP2ndEd/examples/PP2E/Gui/Tour/demoAll_frm.py
demoDlg: none
demoCheck: 1 0 1 1 0
demoRadio: Query
demoScale: 2
```

Единственным хитроумным приемом в этом сценарии является использование встроенной функции Python `__import__` для импорта модуля по строке его имени. Взглянем на следующие две строки из функции сценария `addComponents`:

```
module = __import__(demo)              # импорт модуля по строке имени
part = module.Demo(root)               # прикрепить экземпляр его Demo
```

Это эквивалентно высказываниям типа:

```
import 'demoDlg'
part = 'demoDlg'.Demo(root)
```

за исключением того, что такой синтаксис в Python недопустим – имя модуля в командах импорта должно быть переменной Python, а не строкой. Для обеспечения общности функция `addComponents` проходит по списку строк с именами и с помощью

__import__ импортирует и возвращает модуль, идентифицируемый каждой строкой. Все происходит так, как если бы были выполнены все следующие команды:

```
import demoDlg, demoRadio, demoCheck, demoScale
part = demoDlg.Demo(root)
part = demoRadio.Demo(root)
part = demoCheck.Demo(root)
part = demoScale.Demo(root)
```

Но благодаря тому, что в сценарии использован список строк с именами, легче изменять набор встраиваемых демонстраций – нужно лишь изменить список, не трогая исполняемый код. Кроме того, такой код, управляемый данными, оказывается более компактным, менее избыточным и легче отлаживается и сопровождается. Между прочим, можно также импортировать модули с помощью строк, содержащих их имена, динамически создавая и выполняя команды импорта такого вида:

```
for demo in demoModules:
    exec 'from %s import Demo' % demo        # создать и выполнить from
    part = Demo(root)                        # или eval('Demo')(window)
```

Оператор exec компилирует и выполняет строку с командой Python (в данном случае from для загрузки из модуля класса Demo); оно действует так, как если бы строка с командой была вставлена в исходный код вместо exec. Так как поддерживаются любые команы Python, эта технология является более общей, не ограничиваясь вызовом __import__, но она может замедлять работу, потому что требуется производить синтаксический анализ строк кода перед их выполнением.[1] Но такое замедление может быть несущественным для GUI: пользователи работают медленнее, чем синтаксические анализаторы.

В главе 6 было показано, что такое прикрепление вложенных фреймов является лишь одним из способов повторного использования кода GUI, оформленного в виде классов. Столь же просто организовывать такие интерфейсы путем создания подклассов, а не встраивания. В данном случае, однако, нас больше интересует применение существующего пакета графических элементов, чем его изменение; в следующих двух разделах показаны еще два способа представления таких пакетов пользователям.

Независимые окна

Когда есть набор классов-компонетов, годится любой родительский элемент – и фреймы, и новые окна верхнего уровня. В примере 7.33 все четыре объекта демонстрационных панелей прикрепляются к собственным окнам Toplevel, а не к одному и тому же Frame.

Пример 7.33. PP2E\Gui\Tour\demoAll-win.py

```
########################################################
# 4 демонстрационных класса в независимых окнах верхнего уровня;
# не в процессах: при выходе из одного закрываются все остальные,
# потому что в данном случае все окна выполняются в рамках одного процесса
########################################################

from Tkinter import *
demoModules = ['demoDlg', 'demoRadio', 'demoCheck', 'demoScale']
```

[1] Как будет показано дальше, exec может представлять опасность, если выполняет строки, полученные от других пользователей или по сети. К жестко закодированным строкам данного примера это не относится.

```
demoObjects = []
for demo in demoModules:
    module = __import__(demo)              # импорт по строке имени
    window = Toplevel()                    # создать новое окно
    demo   = module.Demo(window)           # родительским является новое окно
    demoObjects.append(demo)

def allstates():
    for obj in demoObjects:
        if hasattr(obj, 'report'):
            print obj.__module__,
            obj.report()

Label(text='Multiple Toplevel window demo', bg='white').pack()
Button(text='States', command=allstates).pack(fill=X)
mainloop()
```

Мы уже встречались с классом `Toplevel`; каждый его экземпляр создает на экране новое окно. Получаемый результат показан на рис. 7.31 – каждая демонстрация выполняется в собственном окне, а не упаковывается в одно окно.

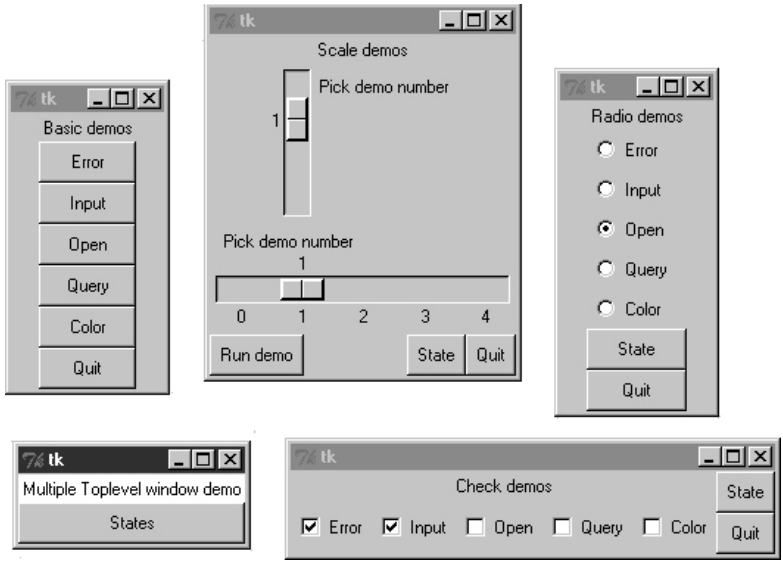


Рис. 7.31. demoAll_win: новые окна Toplevel

Главное корневое окно появляется в левом нижнем углу этого экрана; на нем есть кнопка States, которая запускает метод `report` для каждого объекта демонстрации, выводя в `stdout` такой текст:

```
C:\...\PP2E\Gui\Tour>python demoAll_win.py
in onMove 0
in onMove 0
in onMove 1
you pressed Open
result: C:/PP2ndEd/examples/PP2E/Gui/Tour/demoAll_win.txt
demoRadio Open
demoCheck 1 1 0 0 0
demoScale 1
```

Запуск программ

Наконец, как было показано ранее в этой главе, окна Toplevel функционируют независимо друг от друга, не являясь в действительности отдельными программами. Завершение любого из окон, создаваемых в примере 7.33, завершает их все, потому что все они выполняются в одном и том же программном процессе. В некоторых приложениях это приемлемо, но не во всех.

Полная независимость обеспечивается в примере 7.34, в котором каждая из демонстрационных запускающих программ порождается как независимая с помощью модуля launchmodes, который мы написали в конце главы 3 «Системные средства параллельного выполнения». Это возможно только потому, что все демонстрационные примеры были написаны одновременно как импортируемые классы и выполняемые сценарии, – здесь при их запуске они все получают имя __main__.

Пример 7.34. PP2E\Gui\Tour\demoAll-prg.py

```
########################################################
# 4 демонстрационных класса, запускаемые как независимые программные процессы;
# теперь, когда закрывается одно окно, остальные продолжают жить;
# здесь запустить все вызовы report непросто;
# в некоторых схемах запуска stdout дочерних программ не действует;
########################################################

from Tkinter import *
demoModules = ['demoDlg', 'demoRadio', 'demoCheck', 'demoScale']
from PP2E.launchmodes import PortableLauncher

for demo in demoModules:                        # см. «Системные средства
                                                # параллельного выполнения»
    PortableLauncher(demo, demo+'.py')()        # запуск как программ верхнего уровня

Label(text='Multiple program demo', bg='white').pack()
mainloop()
```

Как видно из рис. 7.32, создаваемый данным сценарием экран аналогичен предыдущему – все четыре демонстрации появляются в собственных окнах. Однако на этот раз окна действительно являются независимыми программами: при завершении любого из пяти имеющихся окон остальные сохраняются.

Обмен данными между программами

Порождение GUI как программ – вершина независимости кода, но затрудняет связь между компонентами. Например, поскольку демонстрационный код выполняется здесь в виде отдельных программ, нет простого способа запустить все их методы report из окна запускающего сценария, показанного в середине рис. 7.32. На самом деле кнопки States теперь нет, и при запуске демонстрационных программ в stdout теперь попадают только сообщения PortableLauncher:

```
C:\...\PP2E\Gui\Tour>python demoAll_prg.py
demoDlg
demoRadio
demoCheck
demoScale
```

На некоторых платформах сообщения, выводимые демонстрационными программами (в том числе собственными кнопками State), могут появиться в исходном окне консоли, в котором запущен сценарий; в Windows вызов os.spawnv, запускающий программы в launchmodes, полностью отключает поток stdout дочерней программы от ро-

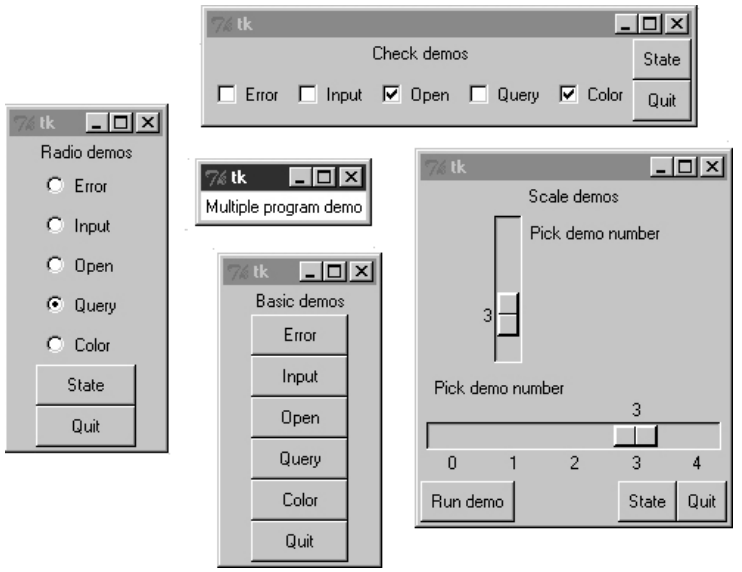


Рис. 7.32. demoAll_prg: независимые программы

дителя. В любом случае невозможно одновременно вызвать методы `report` всех демонстрационных программ одновременно: это порожденные программы в отдельных адресных пространствах, а не импортированные модули.

Конечно, можно запустить методы `report` в порожденных программах с помощью некоторых механизмов IPC, с которыми мы познакомились в главе 3, например:

- Демонстрационные программы могут быть оснащены кодом для приема сигнала от пользователя, в ответ на который запустят свой метод `report`.
- Они могут ждать появления в каналах или очередях строк запросов, посылаемых запускающей программой – запускающая прграмма `demoAll` стала бы фактически клиентом, а демонстрационные GUI – серверами.
- Независимые программы могут общаться подобным образом через сокеты – с этим инструментом мы познакомимся в части III «Создание сценариев для Интернета».

Исходя из управляемой событиями природы программ, основанных на GUI, может потребоваться дополнить их потоками или обратными вызовами по таймеру для периодической проверки таких входящих сообщений в каналах, fifo или сокетах (смотрите, например, метод `after`, описываемый ближе к концу следующей главы). Но поскольку это далеко выходит за рамки данных простых демонстрационных программ, я оставляю такие межпрограммные расширения тем читателям, головы которых устроены более параллельно.

Кодирование, обеспечивающее повторное использование

Постскриптум: я закодировал все демонстрационные панели запуска, разворачиваемые в трех последних примерах, так, чтобы продемонстрировать различные способы, которыми могут быть использованы их графические элементы. Они не были разработаны с учетом общей возможности многократного их использования; на самом деле они не слишком полезны вне контекста знакомства с графическими элементами в данной книге.

Так было задумано; большинством графических элементов Tkinter легко пользоваться после изучения их интерфейсов, а Tkinter в значительной мере сам обеспечивает большую степень гибкости настройки. Но если бы я задумал кодировать классы флажков и переключателей, которые можно повторно использовать как общие библиотечные компоненты, их нужно было бы структурировать иным образом:

Лишние графические элементы

Они не должны выводить ничего, кроме переключателей и флажков. В существующем виде в каждый из демонстрационных примеров для иллюстрации встроены кнопки State и Quit, но в действительности для каждого окна верхнего уровня должна быть одна кнопка Quit.

Управление геометрией

Они должны допускать различное расположение кнопок и вообще не упаковывать себя (или располагать по сетке). В настоящей схеме общецелевого повторного использования часто лучше предоставить вызвавшему управлять геометрией компонентов.

Ограничения режима использования

Они должны либо экспортировать сложные интерфейсы для поддержки всех параметров настройки и режимов Tkinter, либо принять ограничивающие решения, поддерживающие только один общий способ применения. Например, эти кнопки могут выполнять обратные вызовы при нажатии или позволить приложению позже получать их состояние.

В примере 7.35 показан один из путей кодирования панелей флажков и переключателей как библиотечных компонентов. Он основывается на связывании переменных Tkinter и для обеспечения простоты интерфейса требует, чтобы вызывающий использовал общий режим – получение состояния, а не обратные вызовы при нажатии.

Пример 7.35. PP2E\Gui\Tour\buttonbars.py

```
# классы панелей флажков и переключателей для приложений, которые запрашивают состояние позже;
# передать список параметров, вызвать state(), работа с переменными автоматическая

from Tkinter import *

class Checkbar(Frame):
    def __init__(self, parent=None, picks=[], side=LEFT, anchor=W):
        Frame.__init__(self, parent)
        self.vars = []
        for pick in picks:
            var = IntVar()
            chk = Checkbutton(self, text=pick, variable=var)
            chk.pack(side=side, anchor=anchor, expand=YES)
            self.vars.append(var)
    def state(self):
        return map((lambda var: var.get()), self.vars)

class Radiobar(Frame):
    def __init__(self, parent=None, picks=[], side=LEFT, anchor=W):
        Frame.__init__(self, parent)
        self.var = StringVar()
        for pick in picks:
            rad = Radiobutton(self, text=pick, value=pick, variable=self.var)
            rad.pack(side=side, anchor=anchor, expand=YES)
    def state(self):
        return self.var.get()
```

```
if __name__ == '__main__':
    root = Tk()
    lng = Checkbar(root, ['Python', 'C#', 'Java', 'C++'])
    gui = Radiobar(root, ['win', 'x11', 'mac'], side=TOP, anchor=NW)
    tgl = Checkbar(root, ['All'])
    gui.pack(side=LEFT, fill=Y)
    lng.pack(side=TOP,  fill=X)
    tgl.pack(side=LEFT)
    lng.config(relief=GROOVE, bd=2)
    gui.config(relief=RIDGE,  bd=2)
    from quitter import Quitter
    def allstates(): print gui.state(), lng.state(), tgl.state()
    Quitter(root).pack(side=RIGHT)
    Button(root, text='Peek', command=allstates).pack(side=RIGHT)
    root.mainloop()
```

Для повторного использования этих классов в сценариях нужно импортировать их и вызвать со списком параметров, которые должны появиться на панелях флажков или переключателей. Код самотестирования модуля, находящийся в конце, сообщает детали его применения. При выполнении кода как программы, а не при импорте, появляется экран, показанный на рис. 7.33, – окно верхнего уровня с двумя встроенными `Checkbar`, одним `Radiobar`, кнопкой `Quitter` для завершения, а также кнопкой Peek для показа состояния панелей.

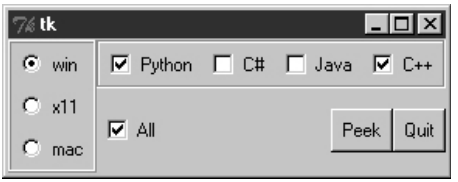


Рис. 7.33. Окно самотестирования buttonbars

В данном случае после нажатия Peek в `stdout` содержится следующий текст – результаты методов `state` этих классов:

```
x11 [1, 0, 1, 1] [0]
win [1, 0, 0, 1] [1]
```

Два класса из этого модуля демонстрируют, насколько просто создаются оболочки интерфейсов Tkinter, облегчающие их использование, – они полностью скрывают многие замысловатые части панелей переключателей и флажков. Например, при использовании таких классов высокого уровня можно полностью забыть о том, как нужно использовать связанные переменные: нужно просто создать объекты со списками параметров, а впоследствии вызвать их методы `state`. Этот путь, если пройти его до конца, приводит к библиотеке графических элементов более высокого уровня, типа пакета `PMW`, о котором говорилось в главе 6.

С другой стороны, эти классы все же не готовы для универсального применения; например, если требуется выполнять действия при нажатии этих кнопок, нужно использовать другие интерфейсы верхнего уровня. К счастью, Python/Tkinter предоставляет их множество. Далее в этой книге мы снова будем пользоваться комбинациями графических элементов и снова применять технологию, с которой познакомились в этой главе, при создании более крупных GUI. А сейчас последняя остановка в этой первой главе, посвященной обзору графических элементов, – фотолаборатория.

Изображения

В Tkinter графические изображения выводятся с помощью создания независимых объектов PhotoImage или BitmapImage и прикрепления их к другим графическим элементам путем установки атрибута image. Кнопки, метки, холсты, текст и меню – все они могут выводить изображения, связывая таким способом готовые графические объекты. Это иллюстрирует пример 7.36, в котором на кнопку помещается картинка.

Пример 7.36. PP2E\Gui\Tour\imgButton.py

```
gifdir = "../gifs/"
from Tkinter import *
win = Tk()
igm = PhotoImage(file=gifdir+"ora-pp.gif")
Button(win, image=igm).pack()
win.mainloop()
```

Более простой пример мне было бы трудно найти: этот сценарий всего лишь создает объект Tkinter PhotoImage для GIF-файла, хранящегося в другом каталоге, и связывает его с параметром image графического элемента Button. Результат показан на рис. 7.34.

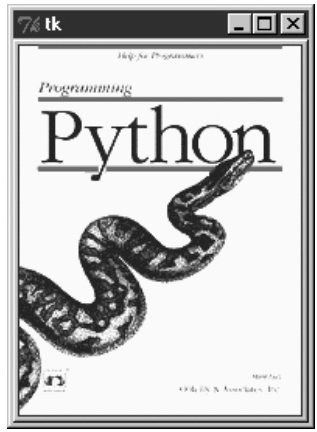


Рис. 7.34. imgButton в действии

PhotoImage и его собрат BitmapImage, в сущности, загружают графические файлы и позволяют прикреплять полученные изображения к другим типам графических элементов. Чтобы открыть файл с картинкой, задайте его имя в атрибуте file этих графических объектов. Графические элементы Canvas – общие окна для вывода графики, подробнее обсуждаемые ниже, тоже могут выводить картинки; пример 7.37 выводит окно, показанное на рис. 7.35.

Пример 7.37. PP2E\Gui\Tour\imgCanvas.py

```
gifdir = "../gifs/"
from Tkinter import *
win = Tk()
img = PhotoImage(file=gifdir+"ora-lp.gif")
can = Canvas(win)
can.pack(fill=BOTH)
can.create_image(2, 2, image=img, anchor=NW)          # координаты x, y
win.mainloop()
```

Размер кнопок автоматически изменяется в соответствии с размером изображения, но размер холста – нет (потому что на холст можно добавлять объекты, как будет показано в главе 8). Для того чтобы размер холста соответствовал размерам изображения, нужно установить их исходя из значений, возвращаемых методами графических объектов `width` и `height`, как в примере 7.38. Эта версия делает при необходимости холст больше или меньше, чем размер, устанавливаемый по умолчанию, разрешает передавать имя графического файла в командной строке и может использоваться в качестве простой утилиты просмотра изображений. Картинка, создаваемая этим сценарием, представлена на рис. 7.36.

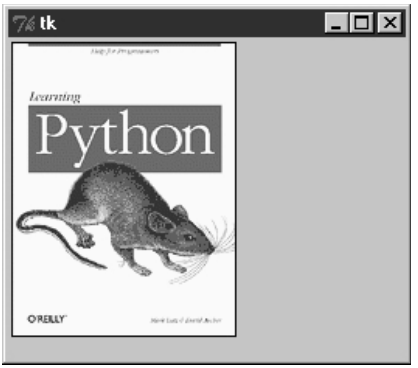


Рис. 7.35. Изображение на холсте

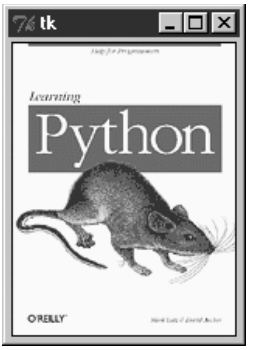


Рис. 7.36. Изменение размера холста соответственно картинке

Пример 7.38. PP2E\Gui\Tour\imgCanvas2.py

```
gifdir = "../gifs/"
from sys import argv
from Tkinter import *
filename = (len(argv) > 1 and argv[1]) or 'ora-lp.gif'     # имя в командной строке?
win = Tk()
img = PhotoImage(file=gifdir+filename)
can = Canvas(win)
can.pack(fill=BOTH)
can.config(width=img.width(), height=img.height())        # размер соответственно картинке
can.create_image(2, 2, image=img, anchor=NW)
win.mainloop()
```

Вот и все. В главе 8 будет показано, как поместить изображение в `Menu`, приведены другие примеры `Canvas` и графический элемент `Text`, дружественный к изображениям. В последующих главах они встретятся в программе показа слайдов (PyView), графическом редакторе (PyDraw), в часах (PyClock), и т. д. В Python/Tkinter добавлять графику в GUI легко.

Однако если есть серьезные намерения заняться использованием графики, можно наткнуться на два подводных камня, о которых я хочу сразу вас предупредить:

Поддерживаемые типы файлов

В настоящее время графический элемент `PhotoImage` поддерживает только файлы форматов GIF, PPM и PGM, а `BitmapImage` поддерживает файлы растровых изображений *.xbm* в стиле X Windows. В последующих версиях количество поддерживаемых форматов может расшириться, и, конечно, можно воспользоваться конвертерами из других форматов в указанные, но более подробное обсуждение этой те-

мы я отложу до врезки «Другие форматы графических файлов: PIL» ниже в этом разделе.

Берегите свои фотографии

В отличие от других графических элементов Tkinter изображение будет утрачено, если соответствующий графический объект Python попадет в уборку мусора. Это значит, что необходимо сохранять явные ссылки на объекты с графикой в течение всего времени, когда они могут понадобиться программе (например, присвоить их переменной с длительным сроком жизни или компоненту структуры данных). Python не сохраняет автоматически ссылку на графическое изображение, даже если оно связано с другими компонентами GUI, отображающими его; кроме того, методы деструкторов изображений стирают их из памяти. Мы уже видели раньше, что переменные Tkinter тоже ведут себя неожиданным образом при уничтожении, но для графики этот эффект еще неприятнее и более вероятен. В будущих версиях Python такое поведение может измениться (хотя есть веские основания не держать вечно в памяти большие графические файлы), но пока элементы с графикой таковы, что если не используются, то могут быть утеряны.

Развлечения с кнопками и картинками

Я хотел представить для этого раздела демонстрацию графики, которая была бы одновременно забавной и полезной. Но ограничился забавностью. Пример 7.39 выводит кнопку, которая случайным образом меняет свою картинку при каждом нажатии.

Пример 7.39. PP2E\Gui\Tour\buttonpics-func.py

```
from Tkinter import *                # получить базовый набор графических элементов
from glob import glob                # список расширения имен файлов
import demoCheck                     # прикрепить к себе демонстрацию флажков
import random                        # случайным образом выбрать картинку
gifdir = '../gifs/'                  # где искать gif-файлы

def draw():
    name, photo = random.choice(images)
    lbl.config(text=name)
    pix.config(image=photo)

root=Tk()
lbl = Label(root,  text="none", bg='blue', fg='red')
pix = Button(root, text="Press me", command=draw, bg='white')
lbl.pack(fill=BOTH)
pix.pack(pady=10)
demoCheck.Demo(root, relief=SUNKEN, bd=2).pack(fill=BOTH)

files = glob(gifdir + "*.gif")                              # gif-файлы на данный момент
images = map(lambda x: (x, PhotoImage(file=x)), files)      # загрузить и держать
print files
root.mainloop()
```

В этом коде используется ряд встроенных инструментов из библиотеки Python:

- Модуль Python `glob`, с которым мы познакомились ранее, дает список всех файлов в каталоге с окончанием *.gif*; иными словами, всех GIF-файлов, которые там хранятся.
- Модуль Python `random` используется для выбора случайного GIF-файла из числа имеющихся в каталоге: `random.choice` случайным образом выбирает и возвращает элемент из списка.

- Чтобы изменить выводимое изображение (и имя GIF-файла, показываемое в метке наверху окна), сценарий просто вызывает метод `config` графического элемента с новыми значениями параметров; такое изменение на лету изменяет отображение графического элемента.

Для разнообразия этот сценарий также прикрепляет экземпляр демонстрационной панели флажков `demoCheck`, который, в свою очередь, прикрепляет экземпляр кнопки `Quitter`, написанной нами ранее. Конечно, это искусственный пример, но он еще раз демонстрирует мощь прикрепления классов компонентов.

Обратите внимание на то, как в этом сценарии создаются все изображения и сохраняются в списке `images`. В данном случае `map` применяет вызов конструктора `PhotoImage` к каждому файлу *.gif* в каталоге с картинками, создавая список наборов `(file,image)`, который сохраняется в глобальной переменной. Помните, что это гарантирует от уборки объектов изображений как мусора в течение времени выполнения программы. На рис. 7.37 показано выполнение этого сценария под Windows.

Рис. 7.37. buttonpics в действии

В этой черно-белой книге можно не заметить, что имя GIF-файла выводится красным по голубому фону метки вверху окна. Окно этой программы автоматически расширяется или сжимается, когда выводятся большие или меньшие GIF-файлы; на рис. 7.38 показано случайно выбранное из каталога графических файлов изображение, имеющее большую высоту.

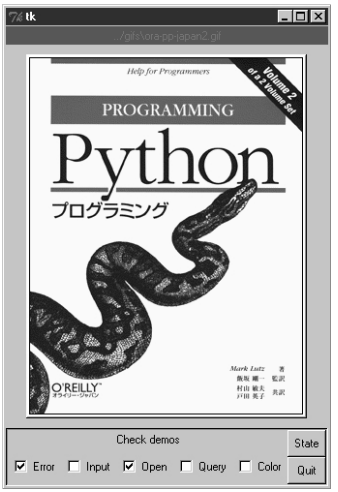


Рис. 7.38. buttonpics, изображающий более высокую картинку

И наконец, на рис. 7.39 показан GUI этого сценария, отображающий один из более широких GIF-файлов, выбранных совершенно случайно из каталога с картинками.[1]

Рис. 7.39. buttonpics обретает политическую направленность

Пока мы забавляемся, перепишем это сценарий в виде класса на случай, если когда-либо в будущем потребуется прикрепить его или перекроить (может произойти и такое). В основном это предполагает создание отступов и добавление `self` перед именами глобальных переменных, как показано в примере 7.40.

Пример 7.40. PP2E\Gui\Tour\buttonpics.py

```
from Tkinter import *                # получить базовый набор графических элементов
from glob import glob                # список расширенных имен файлов
import demoCheck                     # прикрепить к себе пример с флажками
import random                        # выбрать случайную картинку
gifdir = '../gifs/'                  # каталог по умолчанию для загрузки gif-файлов

class ButtonPicsDemo(Frame):
    def __init__(self, gifdir=gifdir, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        self.lbl = Label(self,  text="none", bg='blue', fg='red')
        self.pix = Button(self, text="Press me", command=self.draw, bg='white')
        self.lbl.pack(fill=BOTH)
        self.pix.pack(pady=10)
        demoCheck.Demo(self, relief=SUNKEN, bd=2).pack(fill=BOTH)
        files = glob(gifdir + "*.gif")
        self.images = map(lambda x: (x, PhotoImage(file=x)), files)
        print files

    def draw(self):
        name, photo = random.choice(self.images)
        self.lbl.config(text=name)
        self.pix.config(image=photo)

if __name__ == '__main__': ButtonPicsDemo().mainloop()
```

Эта версия работает так же, как исходная, но теперь ее можно прикрепить к любому другому GUI, в который вы хотите включить такую глупую кнопку.

---

[1] Данная картинка появилась в качестве баннера на сайтах для разработчиков типа *slashdot.com*, когда вышло первое издание «Learning Python». Она породила такую отрицательную реакцию приверженцев Perl, что O'Reilly даже в конце концов убрало объявление. Поэтому, конечно, она оказалась в этой книге.

Другие форматы графических файлов: PIL

Как уже говорилось, сценарии Python Tkinter создают изображения, связывая независимо созданные графические объекты с реальными объектами графических элементов. На момент написания данной книги GUI Tkinter могут выводить графические файлы в форматах GIF, PPM и PGM с помощью объекта `PhotoImage`, а также растровые файлы в стиле X11 (обычно с расширением *.xbm*) с помощью объекта `BitmapImage`.

Множество поддерживаемых форматов файлов ограничено лежащей в основе библиотекой Tk, а не самим Tkinter и может быть расширено в будущем. Но если вам нужно сейчас вывести файлы в другом формате (например, JPEG или BMP), можно либо преобразовать файлы в один из поддерживаемых форматов с помощью программы обработки графики, либо установить пакет расширения Python PIL, о котором говорилось в начале главы 6.

PIL поддерживает в данное время около 30 форматов графических файлов (в том числе JPEG и BMP). Чтобы воспользоваться его средствами, необходимо получить и установить пакет PIL (см. *http://www.pythonware.com*). Затем нужно просто использовать особые объекты `PhotoImage` и `BitmapImage`, импортируемые из модуля PIL `ImageTk`, чтобы открывать файлы в других графических форматах. Это совместимая замена для одноименных классов Tkinter, которую можно использовать везде, где Tkinter предполагает объект `PhotoImage` или `BitmapImage` (то есть в настройках объектов меток, кнопок, холстов, текста и меню). Это означает, что стандартный код Tkinter типа

```
from Tkinter import *
imgobj = PhotoImage(file=imgdir+"spam.gif")
Button(image=imgobj).pack()
```

нужно заменить таким кодом:

```
from Tkinter import *
import ImageTk
imgobj = ImageTk.PhotoImage(file=imgdir+"spam.jpg")
Button(image=imgobj).pack()
```

или в более многословном эквиваленте:

```
from Tkinter import *
import Image, ImageTk
imagefile = Image.open(imgdir+"spam.jpeg")
imageobj  = ImageTk.PhotoImage(imagefile)
Label(image=imageobj).pack()
```

Особенности установки PIL зависят от платформы; на моем переносном компьютере с Windows понадобилось лишь выполнить загрузку, распаковку и добавить каталоги PIL в начало PYTHONPATH. Рассказ о PIL занял бы значительно больше места, чем может позволить эта книга; например, он предоставляет также средства для конвертирования изображений, изменения размеров и преобразования, некоторые из них можно запускать как программы командной строки, не имеющие непосредственного отношения к GUI. Дополнительная информация есть на *http://www.pythonware.com*, а также в наборах электронной документации по PIL и Tkinter.

8

Обзор Tkinter, часть 2

«Меню дня: Spam, Spam и только Spam»

Это вторая глава обзора библиотеки Tkinter, состоящего из двух частей. Она продолжает движение с того места, на котором остановилась глава 7 «Обзор Tkinter, часть 1», и рассказывает о некоторых более сложных графических элементах и инструментах из арсенала Tkinter. В этой главе представлены следующие темы:

- Графические элементы `Menu`, `Menubutton` и `OptionMenu`
- Графический элемент `Scrollbar`: для прокрутки текста, списков и холстов
- Графический элемент `Listbox`: список с вариантами выбора
- Графический элемент `Text`: общее средство вывода и редактирования текста
- Графический элемент `Canvas`: общее средство вывода графики
- Менеджер геометрии `grid`, основанный на таблицах
- Средства, связанные со временем: `after`, `update`, `wait` и потоки
- Основы анимации в Tkinter
- Буферы обмена, удаление графических элементов и окон и т. д.

К концу чтения этой главы вы будете знакомы с основным содержанием библиотеки Tkinter и овладеете всей информацией, необходимой для самостоятельного построения больших переносимых интерфейсов пользователя. Также вы будете готовы справиться с объемными примерами, представленными в главе 9 «Более крупные примеры GUI». В качестве перехода к следующей главе в конце данной будут рассмотрены панели запуска PyDemos и PyGadgets – GUI, используемые для запуска больших примеров GUI.

Меню

Меню представляют собой выпадающие списки, которые вы привыкли видеть в верхней части окна (или всего экрана, если работаете на Macintosh). Переместите курсор мыши на панель меню, щелкните по имени (например, File), и под этим именем появится список вариантов выбора (например, Open, Save). Пункты меню могут запускать какие-либо действия, как при щелчке кнопки; они могут также открывать другие «каскадные» подменю, выводящие дополнительные списки вариантов, показывать окна диалогов и т. д. В Tkinter есть два типа меню, которые можно добавлять в сценарии: меню окон верхнего уровня и меню, основанные на фреймах. Первый вид лучше подходит для окон в целом, а второй может также использоваться в качестве вложенных компонентов.

Меню окон верхнего уровня

В последних версиях Python (с использованием Tk 8.0 и выше) можно связывать горизонтальную панель меню с объектом окна верхнего уровня (например, Tk или Toplevel). В Windows и Unix (X Windows) эта панель меню выводится вдоль верхнего края окна; на Macintosh это меню при выборе окна заменяет то, которое выводится в верхней части экрана. Иными словами, меню окон выглядят так, как принято на той платформе, на которой выполняется сценарий.

В основе этой схемы лежит построение деревьев, состоящих из объектов графических элементов Menu. Нужно просто связать с окном один элемент Menu верхнего уровня, добавить другие объекты выпадающих меню в качестве каскадных для Menu верхнего уровня и добавить записи в каждый из выпадающих списков. Menu перекрестно связываются со следующим более высоким уровнем с помощью аргумента родительского графического элемента и метода add_cascade графического элемента Menu. Это нужно делать так:

1. Создать меню верхнего уровня Menu как дочерний элемент графического элемента окна и задать в качестве атрибута menu окна новый объект Menu.
2. Для каждого выпадающего меню создать новый объект Menu как дочерний для самого верхнего Menu и добавить его как каскадный для самого верхнего Menu с помощью метода add_cascade.
3. Добавить в каждое выпадающее Menu из шага 2 действия, выполняемые при выборе, с помощью параметра command метода add_command, регистрирующего обработчики обратного вызова.
4. Добавить каскадные подменю, создавая новые Menu как дочерние для того объекта Menu, который каскадно расширяется, и связывая родительский и дочерний объекты с помощью add_cascade.

В конечном итоге получается дерево графических элементов Menu с ассоциированными обработчиками обратного вызова command. Однако все это, вероятно, проще показать в коде, чем на словах. В примере 8.1 создается главное меню с двумя выпадающими, File и Edit; у выпадающего меню Edit есть собственное вложенное подменю.

Пример 8.1. PP2E\Gui\Tour\menu_win.py

```
# меню окон верхнего уровня в стиле Tk8.0

from Tkinter import *                           # получить классы графических элементов
from tkMessageBox import *                      # получить стандартные диалоги

def notdone():
    showerror('Not implemented', 'Not yet available')

def makemenu(win):
    top = Menu(win)                             # win-окно верхнего уровня
    win.config(menu=top)                        # установить его параметр menu

    file = Menu(top)
    file.add_command(label='New...',  command=notdone,  underline=0)
    file.add_command(label='Open...', command=notdone,  underline=0)
    file.add_command(label='Quit',    command=win.quit, underline=0)
    top.add_cascade(label='File',     menu=file,        underline=0)

    edit = Menu(top, tearoff=0)
    edit.add_command(label='Cut',     command=notdone,  underline=0)
    edit.add_command(label='Paste',   command=notdone,  underline=0)
    edit.add_separator()
```

```
    top.add_cascade(label='Edit',     menu=edit,        underline=0)

    submenu = Menu(edit, tearoff=0)
    submenu.add_command(label='Spam', command=win.quit, underline=0)
    submenu.add_command(label='Eggs', command=notdone,  underline=0)
    edit.add_cascade(label='Stuff',   menu=submenu,     underline=0)

if __name__ == '__main__':
    root = Tk()                                        # или Toplevel()
    root.title('menu_win')                             # задать информацию для менеджера окон
    makemenu(root)                                     # связать с панелью меню
    msg = Label(root, text='Window menu basics')       # добавить что-нибудь ниже
    msg.pack(expand=YES, fill=BOTH)
    msg.config(relief=SUNKEN, width=40, height=7, bg='beige')
    root.mainloop()
```

В этом файле много кода, устанавливающего обратные вызовы и т. п., поэтому полезно выделить фрагменты, участвующие в процессе построения дерева меню. Для меню File это делается так:

```
top = Menu(win)                              # прикрепить Menu к окну
win.config(menu=top)                         # связать окно с меню
file = Menu(top)                             # прикрепить Menu к Menu верхнего уровня
top.add_cascade(label='File', menu=file)     # связать родительский и дочерний элементы
```

Помимо построения дерева объектов меню этот сценарий демонстрирует некоторые часто встречающиеся параметры конфигурации меню:

Линии-разделители

С помощью `add_separator` сценарий создает в меню Edit разделитель; это просто линия, используемая для разделения групп связанных пунктов.

Линии отрыва

Сценарий запрещает отрыв выпадающего меню Edit, передавая `Menu` параметр `tearoff=0`. Линии отрыва – это пунктирные линии, по умолчанию появляющиеся в верхних меню Tkinter, при щелчке по которым создается новое окно, содержащее меню. Они могут служить удобным упрощающим средством (можно сразу щелкнуть по пункту отрывного меню, не лазая по дереву вложенных пунктов), но не на всех платформах принято их использовать.

Клавиши быстрого вызова

С помощью параметра `underline` уникальная буква в пункте меню становится клавишей быстрого вызова. Параметр задает смещение для буквы быстрого вызова в строке метки пункта меню. Например, в Windows опцию Quit в меню File этого сценария можно выбрать, как обычно, с помощью мыши, а также нажатием клавиши <Alt>, затем «f», затем «q». Использовать `underline` не обязательно – в Windows первая буква имени выпадающего меню автоматически становится клавишей быстрого вызова, а клавиши со стрелками и Enter можно использовать для перемещения и выбора выпадающих пунктов. Но явное задание клавиш может облегчить использование больших меню; например, последовательность клавиш <Alt>+<E>+<S>+<S> выполняет действие по завершению программы во вложенном подменю этого сценария без каких-либо перемещений с помощью мыши или клавиш со стрелками.

Посмотрим, во что все это превращается в переводе на пикселы. На рис. 8.1 показано окно, первоначально появляющееся при запуске этого сценария в Windows; иным, но схожим образом оно выводится в Unix и Macintosh.

Рис. 8.1. menu_win: панель меню окна верхнего уровня

Рис. 8.2 показывает, что происходит при выборе выпадающего меню File. Обратите внимание, что графические элементы `Menu` соединяются, а не упаковываются или располагаются по сетке – в действительности менеджер геометрии здесь не участвует. Если запустить этот сценарий, то можно также заметить, что все элементы меню либо завершают выполнение программы, либо выводят стандартный диалог ошибки «Not Implemented» (не реализовано). Этот пример должен только показать меню, но на практике обработчики событий выбора меню обычно выполняют более полезные вещи.

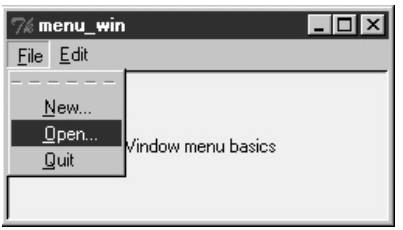


Рис. 8.2. Выпадающее меню File

И наконец, рис. 8.3 показывает, что происходит после щелчка по линии отрыва в меню File и выбора каскадного подменю в выпадающем меню Edit. Каскадные меню можно делать любой глубины вложенности, но злоупотребление этим может не вызвать восторга пользователей.

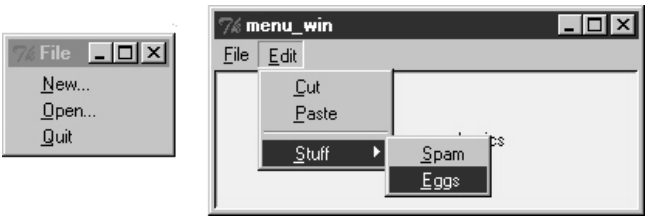


Рис. 8.3. Отрывное меню File и каскадное Edit

В Tkinter меню может иметься во всяком окне верхнего уровня, в том числе во всплывающих окнах, создаваемых графическим элементом `Toplevel`. В примере 8.2 создается три всплывающих окна с той же панелью меню, которую мы только что видели; при запуске создается картина, показанная на рис. 8.4.

Пример 8.2. PP2E\Gui\Tour\menu_win-multi.py

```
from menu_win import makemenu
from Tkinter import *

root = Tk()
```

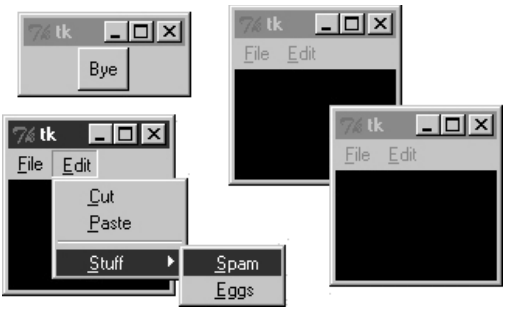


Рис. 8.4. Несколько окон Toplevel с меню

```
for i in range(3):                    # 3 всплывающих окна с меню
    win = Toplevel(root)
    makemenu(win)
    Label(win, bg='black', height=5, width=15).pack(expand=YES, fill=BOTH)
Button(root, text="Bye", command=root.quit).pack()
root.mainloop()
```

Меню, основанные на фреймах и кнопках

В окнах верхнего уровня встречается реже, но допускается создание панели меню как горизонтального `Frame`. Однако прежде чем показывать, как это делается, я хочу объяснить, для чего это нужно. Поскольку такая основанная на фреймах схема не зависит от протоколов окон верхнего уровня, она может применяться для добавления меню в качестве *вложенных* компонентов более крупных интерфейсов. Иными словами, она применима не только для окон верхнего уровня. Например, текстовый редактор PyEdit из главы 9 может использоваться как программа и как прикрепляемый компонент. Мы будем реализовывать выбор в PyEdit с помощью оконных меню при выполнении его как самостоятельной программы, но с помощью меню, основанных на фрейме, когда PyEdit будет встраиваться в экраны PyMail и PyView. Стоит изучить обе схемы.

Для меню, основанных на фреймах, требуется написать несколько дополнительных строчек кода, но они не намного сложнее оконных меню. Для создания такого меню нужно упаковать графические элементы `Menubutton` в контейнере `Frame`, связать графические элементы `Menu` с `Menubutton` и связать `Frame` с верхней частью окна-контейнера. В примере 8.3 создается такое же меню, как в примере 8.2, но с использованием подхода, основанного на фреймах.

Пример 8.3. PP2E\Gui\Tour\menu_frm.py

```
# Меню, использующие фреймы: для окон верхнего уровня и компонентов

from Tkinter import *                           # получить классы графических элементов
from tkMessageBox import *                      # получить стандартные диалоги

def notdone():
    showerror('Not implemented', 'Not yet available')

def makemenu(parent):
    menubar = Frame(parent)                     # relief=RAISED, bd=2...
    menubar.pack(side=TOP, fill=X)

    fbutton = Menubutton(menubar, text='File', underline=0)
    fbutton.pack(side=LEFT)
    file = Menu(fbutton)
```

```
    file.add_command(label='New...',  command=notdone,     underline=0)
    file.add_command(label='Open...', command=notdone,     underline=0)
    file.add_command(label='Quit',    command=parent.quit, underline=0)
    fbutton.config(menu=file)

    ebutton = Menubutton(menubar, text='Edit', underline=0)
    ebutton.pack(side=LEFT)
    edit = Menu(ebutton, tearoff=0)
    edit.add_command(label='Cut',     command=notdone,     underline=0)
    edit.add_command(label='Paste',   command=notdone,     underline=0)
    edit.add_separator()
    ebutton.config(menu=edit)

    submenu = Menu(edit, tearoff=0)
    submenu.add_command(label='Spam', command=parent.quit, underline=0)
    submenu.add_command(label='Eggs', command=notdone,     underline=0)
    edit.add_cascade(label='Stuff',   menu=submenu,        underline=0)
    return menubar

if __name__ == '__main__':
    root = Tk()                                    # или TopLevel, или Frame
    root.title('menu_frm')                         # информация для менеджера окон
    makemenu(root)                                 # связать панель меню
    msg = Label(root, text='Frame menu basics')    # добавить что-нибудь ниже
    msg.pack(expand=YES, fill=BOTH)
    msg.config(relief=SUNKEN, width=40, height=7, bg='beige')
    root.mainloop()
```

Снова выделим здесь логику связывания, чтобы не отвлекали другие детали. Для случая меню File она сводится к следующему:

```
menubar = Frame(parent)                    # создать Frame для menubar
fbutton = Menubutton(menubar, text='File') # прикрепить MenuButton к Frame
file    = Menu(fbutton)                    # прикрепить Menu к MenuButton
fbutton.config(menu=file)                  # связать кнопку с меню
```

В такой схеме появляется дополнительный графический элемент `Menubutton`, но она не намного сложнее создания оконных меню верхнего уровня. На рис. 8.5 и 8.6 показано, как этот сценарий выполняется в Windows.

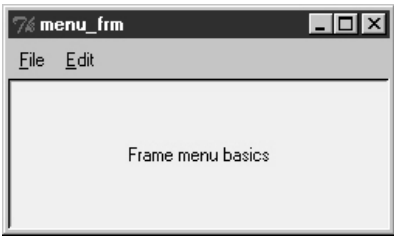


Рис. 8.5. menu_frm: Frame и панель меню Menubutton

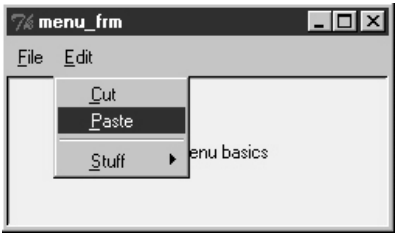


Рис. 8.6. С выбранным меню Edit

Графические элементы меню в этом сценарии по умолчанию предоставляют набор связанных событий, автоматически выводящих меню при выборе мышью. Такие внешний вид и поведение не вполне совпадают с показанной выше схемой оконного меню верхнего уровня, хотя и близки к ней, могут настраиваться теми способами, которые допускают фреймы (то есть с помощью цветов и границ), и будут аналогично выглядеть на всех платформах (что, возможно, является недостатком).

Однако самым большим преимуществом панелей меню, использующих фреймы, является возможность прикрепления в качестве вложенных компонентов к более крупным экранам. Пример 8.4 и создаваемый им интерфейс (рис. 8.7) показывают, каким образом это делается.

Рис. 8.7. Несколько меню Frame в одном окне

Пример 8.4. PP2E\Gui\Tour\menu_frm-multi.py

```
from menu_frm import makemenu          # здесь нельзя использовать menu_win--одно окно,
from Tkinter import *                  # но можно прикреплять импортированные меню к окнам
root = Tk()
for i in range(2):                     # 2 меню вложены в одно окно
    mnu = makemenu(root)
    mnu.config(bd=2, relief=RAISED)
    Label(root, bg='black', height=5, width=15).pack(expand=YES, fill=BOTH)
Button(root, text="Bye", command=root.quit).pack()
root.mainloop()
```

Благодаря отсутствию привязки к охватывающему окну меню, основанные на фреймах, можно использовать в составе другого прикрепляемого компонента графических элементов. Например, режим встраивания меню примера 8.5 действует даже тогда, когда родителем меню является другой контейнер `Frame`, а не окно верхнего уровня.

Пример 8.5. PP2E\Gui\Tour\menu_frm-multi2.py

```
from menu_frm import makemenu          # здесь нельзя использовать menu_win - root=Frame
from Tkinter import *
root = Tk()
for i in range(3):                     # 3 меню, вложенные в контейнеры
    frm = Frame()
    mnu = makemenu(frm)
    mnu.config(bd=2, relief=RAISED)
    frm.pack(expand=YES, fill=BOTH)
    Label(frm, bg='black', height=5, width=15).pack(expand=YES, fill=BOTH)
Button(root, text="Bye", command=root.quit).pack()
root.mainloop()
```

Использование Menubutton и Optionmenu

В действительности меню, основанные на `Menubutton`, являются еще более общими, чем следует из примера 8.3, – фактически они могут появляться в любом месте экра-

на, где может располагаться обычная кнопка, а не только в панели меню Frame. В примере 8.6 создается выпадающий список Menubutton, который показывается самостоятельно и прикреплен к корневому окну; на рис. 8.8 приведен создаваемый им GUI.

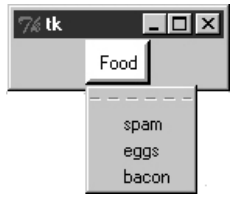


Рис. 8.8. Menubutton сама по себе

Пример 8.6. PP2E\Gui\Tour\mbutton.py

```
from Tkinter import *
root    = Tk()
mbutton = Menubutton(root, text='Food')     # выпадающее меню располагается отдельно
picks   = Menu(mbutton)
mbutton.config(menu=picks)
picks.add_command(label='spam',  command=root.quit)
picks.add_command(label='eggs',  command=root.quit)
picks.add_command(label='bacon', command=root.quit)
mbutton.pack()
mbutton.config(bg='white', bd=4, relief=RAISED)
root.mainloop()
```

Родственный графический элемент Tkinter Optionmenu выводит элемент, выбранный в выпадающем меню. Он похож на Menubutton, к которому добавлена метка, и при щелчке по нему выводит меню вариантов выбора; однако для того чтобы получить результат выбора, нужно не регистрировать обратный вызов, а связывать переменные Tkinter (описанные в главе 7); элементы меню передаются в конструктор в виде аргументов вслед за переменной.

Пример 8.7 иллюстрирует типичное использование Optionmenu и строит интерфейс, показанный на рис. 8.9. Щелчок по любой из первых двух кнопок открывает выпадающее меню; щелчок по третьей кнопке «state» получает и выводит текущие значения, отображенные на первых двух.

Рис. 8.9. Optionmenu в действии

Пример 8.7. PP2E\Gui\Tour\optionmenu.py

```
from Tkinter import *
root = Tk()
var1 = StringVar()
var2 = StringVar()
opt1 = OptionMenu(root, var1, 'spam', 'eggs',  'toast')     # как Menubutton
opt2 = OptionMenu(root, var2, 'ham',  'bacon', 'sausage')   # но показывает выбранное значение
```

```
opt1.pack(fill=X)
opt2.pack(fill=X)
var1.set('spam')
var2.set('ham')
def state(): print var1.get(), var2.get()                    # связанные переменные
Button(root, command=state, text='state').pack()
root.mainloop()
```

Есть и другие относящиеся к меню темы, которые мы пропустим из-за недостатка места. Например, сценарии могут добавлять элементы в *системные меню* и создавать *всплывающие меню* (показываемые в ответ на события, не будучи связанными с кнопкой). Подробнее на эту тему читайте в ресурсах Tk и Tkinter.

Кроме простых пунктов и каскадов в меню могут быть отключенные пункты, флажки и переключатели, графические образы. В следующем разделе показывается, как программируются некоторые из этих специальных пунктов меню.

Окна, содержащие меню и панель инструментов

Помимо меню, выводимого в верхней части, окна часто выводят ряд кнопок в нижней части. Этот нижний ряд кнопок обычно называют панелью инструментов (*toolbar*), и он часто содержит средства для ускоренного выбора пунктов, присутствующих в верхнем меню. В Tkinter добавить в окно панель инструментов просто: нужно упаковать кнопки (и графические элементы других типов) во фрейм, упаковать фрейм в нижнюю часть окна и задать его расширение только в горизонтальном направлении. Это просто очередная реализация иерархической структуры GUI, однако нужно следить за тем, чтобы панели инструментов (и основанные на фреймах панели меню) упаковывались на ранних стадиях, чтобы когда окно сжимается, сначала обрезались графические элементы, находящиеся в середине экрана.

В примере 8.8 показан один из способов добавления в окно панели инструментов. Он демонстрирует также, как можно добавить графику в пункты меню (установить в атрибуте `image` объект `PhotoImage`) и как отключать пункты меню, изображая их при этом серыми (вызвать метод меню `entryconfig`, передав ему индекс отключаемого пункта, отсчитываемый от 1). Заметьте, что объекты `PhotoImage` сохраняются в виде списка; помните, что в отличие от других графических элементов, они будут утеряны, если их не удерживать.

Пример 8.8. PP2E\Gui\Tour\menuDemo.py

```
#!/usr/local/bin/python
###############################################################################
# Меню главного окна в стиле Tk8.0. Панели меню/инструментов пакуются прежде середины,
# fill=X (первым упакован - последним урезан); добавляет элементы меню с графикой;
# см. также: add_checkbutton, add_radiobutton
###############################################################################
from Tkinter import *                           # получить классы графических элементов
from tkMessageBox import *                      # получить стандартные диалоги
class NewMenuDemo(Frame):                       # расширенный фрейм
    def __init__(self, parent=None):            # прикрепить к корневому окну?
        Frame.__init__(self, parent)            # init для суперкласса
        self.pack(expand=YES, fill=BOTH)
        self.createWidgets()                    # прикрепить фреймы/графические элементы
        self.master.title("Toolbars and Menus")  # инфо для менеджера окон
        self.master.iconname("tkpython")        # метка при свертывании
    def createWidgets(self):
        self.makeMenuBar()
```

```
        self.makeToolBar()
        L = Label(self, text='Menu and Toolbar Demo')
        L.config(relief=SUNKEN, width=40, height=10, bg='white')
        L.pack(expand=YES, fill=BOTH)
    def makeToolBar(self):
        toolbar = Frame(self, cursor='hand2', relief=SUNKEN, bd=2)
        toolbar.pack(side=BOTTOM, fill=X)
        Button(toolbar, text='Quit',  command=self.quit    ).pack(side=RIGHT)
        Button(toolbar, text='Hello', command=self.greeting).pack(side=LEFT)
    def makeMenuBar(self):
        self.menubar = Menu(self.master)
        self.master.config(menu=self.menubar)    # master - это окно верхнего уровня
        self.fileMenu()
        self.editMenu()
        self.imageMenu()
    def fileMenu(self):
        pulldown = Menu(self.menubar)
        pulldown.add_command(label='Open...', command=self.notdone)
        pulldown.add_command(label='Quit',    command=self.quit)
        self.menubar.add_cascade(label='File', underline=0, menu=pulldown)

    def editMenu(self):
        pulldown = Menu(self.menubar)
        pulldown.add_command(label='Paste',   command=self.notdone)
        pulldown.add_command(label='Spam',    command=self.greeting)
        pulldown.add_separator()
        pulldown.add_command(label='Delete',  command=self.greeting)
        pulldown.entryconfig(4, state=DISABLED)
        self.menubar.add_cascade(label='Edit', underline=0, menu=pulldown)
    def imageMenu(self):
        photoFiles = ('guido.gif', 'pythonPowered.gif', 'ppython_sm_ad.gif')
        pulldown = Menu(self.menubar)
        self.photoObjs = []
        for file in photoFiles:
            img = PhotoImage(file='../gifs/' + file)
            pulldown.add_command(image=img, command=self.notdone)
            self.photoObjs.append(img)   # сохранить ссылку
        self.menubar.add_cascade(label='Image', underline=0, menu=pulldown)
    def greeting(self):
        showinfo('greeting', 'Greetings')
    def notdone(self):
        showerror('Not implemented', 'Not yet available')
    def quit(self):
        if askyesno('Verify quit', 'Are you sure you want to quit?'):
            Frame.quit(self)
if __name__ == '__main__':  NewMenuDemo().mainloop()  # если выполняется как сценарий
```

При выполнении этого сценария сначала генерируется картина, приведенная на рис. 8.10. Рис. 8.11 показывает это окно после того, как оно несколько растянуто, меню File и Edit оторваны, а меню Image выбрано. В третьем меню сценария – создатель Python Гвидо ван Россум (в очках, объявленных теперь устаревшими). Запустите сценарий на своем компьютере, чтобы лучше почувствовать, как он работает.[1]

[1] Обратите внимание, что элементы панели управления тоже могут быть картинками – просто свяжите с кнопками панели управления маленькие картинки, как показано в конце главы 7.

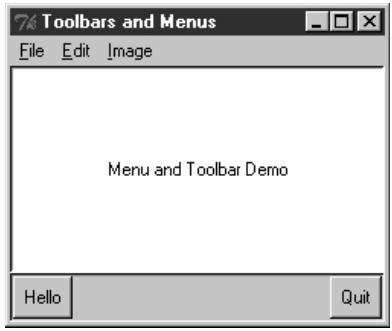


Рис. 8.10. menuDemo: меню и панели инструментов

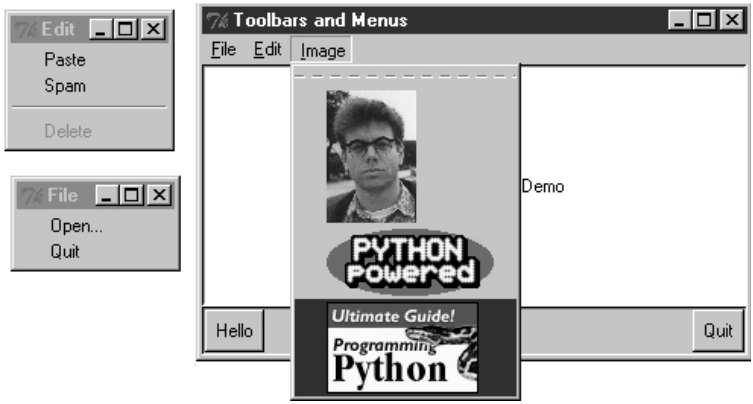


Рис. 8.11. Графика и отрывные меню

Автоматизация создания меню

Меню служат мощным инструментом создания интерфейса в Tkinter. Однако для таких, как я, примеры этого раздела кажутся слишком трудоемкими. При непосредственном вызове методов Tkinter требуется писать много кода, и велика вероятность ошибок. Лучше было бы автоматически строить и связывать меню, описывая их содержимое на более высоком уровне. Мы действительно так и сделаем – в главе 9 мы познакомимся с инструментом, носящим название *GuiMixin*, который автоматизирует процесс создания меню исходя из заданной структуры данных, описывающей все необходимые меню. Дополнительным преимуществом является поддержка меню в обоих стилях – оконном и фреймовом, благодаря чему они могут использоваться как в самостоятельных программах, так и во вложенных компонентах. Важно помнить, какие вызовы лежат в основе создания меню, но не обязательно запоминать их навсегда.

Окна списков и полосы прокрутки

Графические элементы `Listbox` позволяют выводить список элементов, из которых может производиться выбор, а элемент `Scrollbar` предназначается для прокрутки содержимого других графических элементов. Эти графические элементы часто используются совместно, поэтому будем изучать тот и другой одновременно. В примере 8.9 `Listbox` и `Scrollbar` создаются в виде упакованного набора.

Пример 8.9. PP2E\Gui\Tour\scrolledlist.py

```
from Tkinter import *
class ScrolledList(Frame):
    def __init__(self, options, parent=None):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)                  # сделать расширяемым
        self.makeWidgets(options)
    def handleList(self, event):
        index = self.listbox.curselection()               # при двойном щелчке по списку
        label = self.listbox.get(index)                   # получить выбранный текст
        self.runCommand(label)                            # и вызвать действие
    def makeWidgets(self, options):                       # или get(ACTIVE)
        sbar = Scrollbar(self)
        list = Listbox(self, relief=SUNKEN)
        sbar.config(command=list.yview)                   # перекрестная ссылка sbar и list
        list.config(yscrollcommand=sbar.set)              # сдвиг одного => сдвиг другого
        sbar.pack(side=RIGHT, fill=Y)                     # первым упакован - последним обрезан
        list.pack(side=LEFT, expand=YES, fill=BOTH)       # первым обрезается список
        pos = 0
        for label in options:                             # добавить в окно списка
            list.insert(pos, label)                       # или insert(END,label)
            pos = pos + 1
       #list.config(selectmode=SINGLE, setgrid=1)         # режимы выбора, изменения размера
        list.bind('<Double-1>', self.handleList)          # установить обработчик события
        self.listbox = list
    def runCommand(self, selection):                      # переопределить ниже
        print 'You selected:', selection
if __name__ == '__main__':
    options = map((lambda x: 'Lumberjack-' + str(x)), range(20))
    ScrolledList(options).mainloop()
```

Этот модуль можно выполнить автономно, чтобы поэкспериментировать с данными графическими элементами, но он может также использоваться в качестве библиотечного объекта. Передавая различные списки выбора в аргумент `options` и переопределяя метод `runCommand` в подклассе, можно повторно использовать определенный здесь класс компонента `ScrolledList` при каждой необходимости вывести список с прокруткой. Слегка поразмыслив, можно легко расширить библиотеку Tkinter классами Python.

При автономном выполнении этот сценарий создает окно, показанное на рис. 8.12. Это `Frame` с `Listbox` в левой части, содержащим 20 сгенерированных элементов (щелчок сделан по пятому) и связанным с `Scrollbar` в правой части для перемещения по списку. Если переместить ползунок в полосе прокрутки, перемещается список, и наоборот.

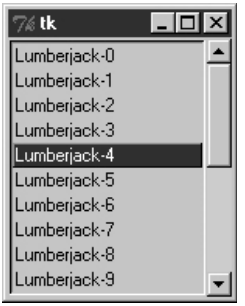


Рис. 8.12. scrolledlist вверху

Программирование окон списков

Окна списков использовать просто, но заполняются и обрабатываются они довольно своеобразными способами в сравнении с теми графическими элементами, которые рассматривались до сих пор. В обращения к окнам списков часто передается *индекс*, указывающий на элемент списка. Индексы нумеруются целыми числами начиная с 0, но Tkinter принимает также вместо чисел смещений особые строки имен – «end» для ссылки на конец списка, «active» для обозначения выбранной строки и другие. Поэтому обычно программировать обращение к окну списка можно несколькими способами.

Например, следующий сценарий добавляет элементы к списку, находящемуся в окне, вызывая его метод `insert` и последовательно увеличивая смещение (начиная с нуля):

```
list.insert(pos, label)
pos = pos + 1
```

Но список можно заполнить и добавляя элементы в его конец, вообще не ведя счетчика положения, с помощью любого из следующих предложений:

```
list.insert('end', label)     # добавление в конец: отсчет позиций не нужен
list.insert(END, label)       # END устанавливается в Tkinter равным 'end'
```

У графического элемента `listbox` нет чего-либо подобного параметру `command`, с помощью которого регистрируются обработчики обратных вызовов для нажатий кнопок, поэтому необходимо получать выделенные в нем элементы во время обработки событий других графических элементов (например, нажатий кнопок в GUI) либо обрабатывать сделанные пользователем выделения, внедряясь в протоколы других событий. В следующем сценарии для получения выбранного значения событие `<Double-1>` двойного щелчка левой кнопкой мыши связывается с методом обработчика обратного вызова с помощью `bind` (рассмотренного ранее в этом обзоре).

В обработчике двойного щелчка этот сценарий получает выделенный в окне списка элемент с помощью такой пары методов:

```
index = self.listbox.curselection()       # получить индекс выделенного элемента
label = self.listbox.get(index)           # получить текст, соответствующий этому индексу
```

Здесь тоже можно по-разному написать код. Обе следующие строки дают одинаковый результат: они получают содержимое строки с индексом «active», то есть выбранной в данный момент:

```
label = self.listbox.get('active')        # получить по индексу active
label = self.listbox.get(ACTIVE)          # в Tkinter ACTIVE='active'
```

Для иллюстрации метод класса по умолчанию `runCommand` выводит выбранное значение при каждом двойном щелчке по элементу списка – сценарий возвращает его в виде строки с текстом выбранного элемента:

```
C:\...\PP2E\Gui\Tour>python scrolledlist.py
You selected: Lumberjack-2
You selected: Lumberjack-19
You selected: Lumberjack-4
You selected: Lumberjack-12
```

Программирование полос прокрутки

Однако самое большое таинство в этом сценарии сводится к двум строкам кода:

```
sbar.config(command=list.yview)             # вызвать list.yview при перемещении
list.config(yscrollcommand=sbar.set)        # вызвать sbar.set при перемещении
```

С помощью этих параметров настройки полоса прокрутки и окно списка обоюдно связываются между собой: их значения просто ссылаются на связанные методы друг друга. Благодаря такому соединению Tkinter автоматически синхронизирует два графических элемента при перемещениях в них. Вот как это действует:

- Перемещение полосы прокрутки вызывает обработчик обратного вызова, зарегистроированный с помощью ее параметра `command`. Здесь `list.yview` ссылается на встроенный метод окна списка, который пропорционально настраивает отображение окна списка, исходя из аргументов, переданных обработчику.
- При вертикальном перемещении в окне списка вызывается обработчик обратного вызова, зарегистрированный в его параметре `yscrollcommand`. В данном сценарии встроенный метод `sbar.set` пропорционально настраивает полосу прокрутки.

Иными словами, при движении в одном элементе автоматически осуществляется движение в другом. В Tkinter у всех прокручиваемых элементов – `Listbox`, `Entry`, `Text` и `Canvas` – есть встроенные методы `yview` и `xview` для обработки прокрутки по вертикали и по горизонтали в поступающих обратных вызовах, а также параметры `yscrollcommand` и `xscrollcommand` для задания обработчика обратного вызова связанной с ними полосы прокрутки. У полос прокрутки есть параметр `command`, в котором указывается обработчик, вызываемый при перемещении. Tkinter внутри себя передает этим методам информацию, задающую их новое положение (например, «спуститься сверху на 10% »), но в сценариях программисту не требуется опускаться до таких деталей.

Так как полоса прокрутки и окно списка взаимно связаны путем установки их параметров, при перемещении полосы прокрутки автоматически происходит перемещение в списке, а при перемещении в списке автоматически перемещается полоса прокрутки. Для перемещения полосы прокрутки нужно перетащить ее рельефную часть либо щелкнуть по стрелке или пустой области. Для перемещения в списке щелкните по нему и переместите курсор мыши выше или ниже окна, не отпуская кнопки мыши. В обоих случаях список и полоса прокрутки двигаются в унисон. На рис. 8.13 показано, что произойдет после перемещения в списке на несколько элементов вниз тем или иным способом.

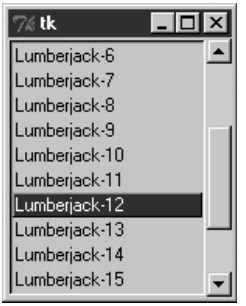


Рис. 8.13. scrolledlist в середине

Упаковка полос прокрутки

Наконец, вспомним, что графические элементы, упаковываемые последними, всегда обрезаются первыми при уменьшении размеров окна. По этой причине важно упаковать полосы прокрутки как можно раньше, чтобы они исчезли последними, когда окно уменьшится до таких размеров, что в нем нельзя будет ничего показать. Обычно можно справиться с тем, что окно списка выведено не полностью, но полоса прокрут-

ки необходима для перемещения по списку. Как показано на рис. 8.14, при уменьшении окна сценария отрезается часть списка, но полоса прокрутки сохраняется.

Рис. 8.14. scrolledlist становится маленьким

В то же время, обычно не требуется, чтобы полоса прокрутки расширялась вместе с окном, поэтому упаковывайте ее просто с `fill=Y` (или `fill=X` для прокрутки по горизонтали), но не с `expand=YES`. В частности, расширение окна этого примера увеличивает окно списка, но сохраняет полосу прокрутки прикрепленной справа с той же шириной.

В примерах этой и последующих глав мы неоднократно будем встречаться с полосами прокрутки и окнами списков (можно заглянуть вперед и посмотреть примеры PyEdit, PyForm, PyTree и ShellGui). И хотя основы применения схвачены в данной главе, следует отметить, что не все, касающееся того и другого графического элемента, попалось нам здесь на глаза.

Например, столь же легко к прокручиваемым графическим элементам можно добавить *горизонтальные* полосы прокрутки; они программируются почти так же, как вертикальные, но имена обработчиков начального вызова начинаются с «x», а не «y», а для объекта полосы прокрутки устанавливается параметр `orient='horizontal'` (примеры смотрите дальше в программах PyEdit и PyTree). Окна списков могут быть полезны в качестве средств ввода данных, даже если к ним не прикрепляются полосы прокрутки; они также принимают параметры, устанавливающие цвет, шрифт и рельеф, и поддерживают выбор нескольких пунктов (по умолчанию устанавливается `selectmode=SINGLE`, определяющий выбор только одного пункта).

Полосы прокрутки тоже могут действовать в GUI различными способами – их можно связывать с графическими элементами других типов из библиотеки Tkinter. Например, их часто прикрепляют к элементу `Text`, что приводит нас к следующей теме данного обзора.

Text

Уже отмечалось, что самыми мощными в Tkinter являются графические элементы текста и холста. Оба они обладают богатым набором функций. Например, графический элемент Tkinter `Text` оказался достаточно мощным для реализации веб-броузера Grail, обсуждаемого в главе 15 «Более сложные темы Интернета»: он поддерживает сложные установки стилей шрифтов, встраивание графики и многое другое. Графический элемент Tkinter `Canvas` (средство общего назначения для отображения графики) также положен в основу многих приложений для сложной обработки изображений и визуализации.

В главе 9 мы воспользуемся двумя этими графическими элементами для реализации текстового редактора (PyEdit), графического редактора (PyDraw), GUI-часов (PyClock) и программы для просмотра слайд-шоу (PyView). Однако эту главу мы начнем с применения этих графических элементов в более простых примерах. В примере 8.10 реализовано простое средство отображения текста с прокруткой, которое может вывести строку текста или файл.

Пример 8.10. PP2E\Gui\Tour\scrolledtext.py

```
# простой компонент для просмотра текста или файла

print 'PP2E scrolledtext'
from Tkinter import *

class ScrolledText(Frame):
    def __init__(self, parent=None, text='', file=None):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)                # сделать расширяемым
        self.makewidgets()
        self.settext(text, file)
    def makewidgets(self):
        sbar = Scrollbar(self)
        text = Text(self, relief=SUNKEN)
        sbar.config(command=text.yview)                 # взаимно связать sbar и text
        text.config(yscrollcommand=sbar.set)            # перемещение одного перемещает другой
        sbar.pack(side=RIGHT, fill=Y)                   # пакуется первым - обрезается последним
        text.pack(side=LEFT, expand=YES, fill=BOTH)     # сначала обрезается text
        self.text = text
    def settext(self, text='', file=None):
        if file:
            text = open(file, 'r').read()
        self.text.delete('1.0', END)                    # удалить текущий текст
        self.text.insert('1.0', text)                   # добавить в строке 1, колонке 0
        self.text.mark_set(INSERT, '1.0')               # установить курсор вставки
        self.text.focus()                               # сэкономить щелчок мыши
    def gettext(self):                                  # возвращает строку
        return self.text.get('1.0', END+'-1c')          # с первой до последней

if __name__ == '__main__':
    root = Tk()
    try:
        st = ScrolledText(file=sys.argv[1])             # имя файла в командной строке
    except IndexError:
        st = ScrolledText(text='Words\ngo here')        # или нет: 2 строчки
    def show(event): print repr(st.gettext())           # показ необработанной строки
    root.bind('<Key-Escape>', show)                     # esc = дамп текста
    root.mainloop()
```

Подобно `ScrolledList` примера 8.9, объект `ScrolledText` в этом файле разработан как многократно используемый компонент, но может выполняться автономно, выводя содержимое текстового файла. И так же, как в предыдущем разделе, этот сценарий сначала упаковывает полосу прокрутки, чтобы она исчезла с экрана последней при сжатии окна, и настраивает встроенный объект `Text` для расширения в обоих направлениях при увеличении размеров окна. При запуске с именем файла в качестве аргумента этот сценарий создает окно, показанное на рис. 8.15: он встраивает графический элемент `Text` в левую часть окна и взаимосвязанную с ним полосу прокрутки – в правую.

Для забавы я заполнил текстовый файл, выводимый в окне, с помощью следующего кода и командных строк:

```
C:\...\PP2E\Gui\Tour>type temp.py
f = open('temp.txt', 'w')
for i in range(250):
    f.write('%03d)  All work and no play makes Jack a dull boy.\n' % i)
f.close()

C:\...\PP2E\Gui\Tour>python temp.py
```

Рис. 8.15. scrolledtext в действии

```
C:\...\PP2E\Gui\Tour>python scrolledtext.py temp.txt
PP2E scrolledtext
```

Для просмотра файла передайте его имя в командной строке – текст файла будет автоматически выведен в новом окне. По умолчанию он выводится шрифтом переменной ширины, но в следующем примере мы изменим это, передав графическому элементу параметр `font`.

Обратите внимание на сообщение «PP2E scrolledtext», выводимое при выполнении этого сценария. Поскольку в стандартном дистрибутиве Python есть также файл *ScrolledText.py* с совершенно иным интерфейсом, данный сценарий идентифицирует себя при выполнении или импортировании, чтобы можно было отличить один от другого. Если стандартный сценарий будет исключен из дистрибутива, импортируйте тот, который здесь приведен, чтобы получить простое средство просмотра текста, и настройте конфигурирующие вызовы, чтобы они содержали спецификатор «.text» (библиотечная версия создает подкласс `Text`, а не `Frame`).

Программирование графического элемента Text

Чтобы понять, как вообще работает этот сценарий, придется разобраться с некоторыми деталями графического элемента `Text`. Мы уже знакомы с графическими элементами `Entry` и `Message`, которые обеспечивают некоторое подмножество возможных применений элемента `Text`. Графический элемент `Text` значительно богаче функциями и интерфейсами – он поддерживает ввод и отображение нескольких строк текста, операции редактирования для программ и интерактивных пользователей, различные шрифты и цвета, а также многое другое. Объекты `Text` создаются, настраиваются и упаковываются так же, как любые другие графические элементы, но у них есть особые свойства.

Текст является строкой Python

Хотя графический элемент Text является мощным средством, его интерфейс можно свести к двум базовым понятиям. Во-первых, содержимое элемента Text представляется в сценариях Python в виде строки, и несколько строчек разделяются обычным терминатором \n. Например, строка 'Words\ngo here' представляет две строчки при записи в элемент Text или получении из него; обычно у нее есть и замыкающий \n, но это не обязательно.

Чтобы проиллюстрировать это, данный сценарий связывает нажатие клавиши Escape с получением и выводом всего содержимого находящегося в нем графического элемента Text:

```
C:\...\PP2E\Gui\Tour>python scrolledtext.py
PP2E scrolledtext
'Words\012go here'
'Always look\012on the bright\012side of life\012'
```

При запуске с аргументами этот сценарий записывает в текстовый элемент все содержимое файла. При запуске без аргументов сценарий вставляет в графический элемент простую литеральную строку, выведенную при первом нажатии Escape (вспомните, что \012 является восьмеричным escape-кодом терминатора \n). Второй вывод получен при нажатии Escape в уменьшенном окне, показанном на рис. 8.16.

Рис. 8.16. scrolledtext учится смотреть на жизнь позитивно[1]

Позиция в строке

Вторым ключевым моментом в понимании кода Text является задание *позиции* в строке текста. Как и окно списка, текстовый графический элемент позволяет задавать эту позицию различными способами. Методы Text, предполагающие указание позиции, принимают индекс (*index*), метку (*mark*) или тег (*tag*). Кроме того, некоторые особые операции вызываются с предопределенными метками или тегами – курсор вставки имеет метку INSERT, а текущее выделение имеет тег SEL.

Индексы в Text. Будучи многострочным графическим элементом, Text поддерживает индексы строк и колонок. Например, рассмотрим интерфейсы базовых операций ввода, удаления и получения текста, используемые в этом сценарии:

```
self.text.insert('1.0', text)              # вставить текст в начало
self.text.delete('1.0', END)               # полностью удалить текущий текст
return self.text.get('1.0', END+'-1c')     # получить текст с начала до конца
```

Во всех этих предложениях первый аргумент является абсолютным *индексом*, ссылающимся на начало строки текста: строка «1.0» означает ряд 1, колонку 0 (строки нумеруются с 1, а колонки – с 0). Индекс «2.1» ссылается на второй символ во втором ряду.

Как и у окна списка, для индексов текстового элемента есть символические имена: END в предшествующем вызове delete указывает на позицию, находящуюся сразу за

[1] Текст в окне: «Всегда смотри на светлую сторону жизни». – *Примеч. ред.*

последним символом в строке текста (это переменная Tkinter, имеющая предустановленное строковое значение «end»). Аналогично символический индекс INSERT (на самом деле строка «insert») ссылается на позицию, находящуюся сразу за курсором вставки – то место, где будут появляться символы, вводимые с клавиатуры. Символические имена типа INSERT можно также назвать метками, которые будут описаны несколько далее.

Дополнительной точности можно достичь, добавляя к строкам индексов простые арифметические расширения. Индексное выражение END+'-1c' в вызове get предыдущего примера в действительности является строкой «end-1c» и ссылается на позицию, находящуюся на один символ раньше, чем END. Поскольку END указывает на место сразу за последним символом строки текста, это выражение ссылается на сам последний символ. В результате расширение -1c отрезает замыкающий \n, который графический элемент добавляет к своему содержимому (и который может добавить пустую строку при сохранении в файле).

Аналогичные расширения строк индексов позволяют задавать символы, находящиеся впереди (+1c), строки, находящиеся позади и впереди (+2l, -2l), и задавать такие вещи, как начало слова или строки, в которых находится индекс (lineend, wordstart). Индексы присутствуют в большинстве вызовов для графического элемента Text.

Метки Text. Помимо строк-идентификаторов ряд/колонка можно передавать позиции как имена меток – символические имена позиций между двумя символами. В отличие от абсолютных позиций ряда/колонки метки являются виртуальными адресами, которые перемещаются при вставке нового текста или его удалении, выполняемых сценарием или пользователем. Метка всегда ссылается на первоначальное местоположение, даже если оно смещается с течением времени в другой ряд или колонку.

Чтобы создать метку, вызовите метод text mark_set со строкой имени и индексом, задающим ее логическое положение. Например, в следующем сценарии курсор вставки первоначально устанавливается в начало текста с помощью первого из вызовов:

```
self.text.mark_set(INSERT, '1.0')            # установить курсор вставки в начало
self.text.mark_set('linetwo', '2.0')         # пометить текущую строку 2
```

Имя INSERT является специальной предустановленной меткой, идентифицирующей позицию курсора вставки. Чтобы создать собственную метку, передайте уникальное имя, как во втором вызове, и используйте там, где требуется задать позицию в тексте. Вызов mark_unset удаляет метку по имени.

Теги Text. Кроме абсолютных индексов и символических имен меток, графический элемент Text поддерживает понятие *тегов* – символических имен, ассоциируемых с одной или несколькими подстроками в строке графического элемента Text. Теги можно применять с разными целями, в том числе для представления позиции там, где это требуется: помеченные тегами элементы задаются индексами начала и конца, которые можно впоследствии передавать вызовам, требующим указания позиции.

Например, Tkinter предоставляет встроенный тег с именем SEL – имя Tkinter с предопределенным строковым значением «sel», – которое автоматически ссылается на текст, выделенный в данный момент. Чтобы получить текст, выделенный (подсвеченный) с помощью мыши, выполните любой из следующих вызовов:

```
text = self.text.get(SEL_FIRST, SEL_LAST)      # теги для индексов от/до
text = self.text.get('sel.first', 'sel.last')  # строки и константы тоже действуют
```

Имена SEL_FIRST и SEL_LAST относятся просто к переменным модуля Tkinter с предустановленными значениями, ссылающимися на строки из второго предложения. Метод get предполагает задание двух индексов; для получения текста по тегу добавьте к его имени .first и .last и получите индексы начала и конца.

Чтобы пометить тегом подстроку, вызовите метод элемента tag_add, передав ему строку имени тега и позиции начала и конца (можно помечать тегами текст, добавляемый в вызовах insert). Чтобы снять тег со всех символов в некотором диапазоне символов, вызовите tag_remove:

```
self.text.tag_add('alltext', '1.0', END)  # пометить тегом весь текст элемента
self.text.tag_add(SEL, index1, index2)    # выделить от index1 до index2
self.text.tag_remove(SEL, '1.0', END)     # снять выделение с текста
```

Здесь в первой строке создается новый тег для всего текста в графическом элементе – от начальной до конечной позиции. Во второй строке диапазон символов добавляется во встроенный тег выделения SEL – эти символы автоматически подсвечиваются, поскольку для этого тега предопределена такая настройка его элементов. В третьей строке все символы строки текста удаляются из тега SEL (снимаются все выделения). Обратите внимание, что вызов tag_remove просто снимает тег с текста в указанном диапазоне; для того чтобы полностью удалить тег, нужно вызвать tag_delete.

Можно также динамически отображать индексы в теги. Например, метод текста search возвращает индекс row.column первого вхождения строки между начальной и конечной позициями. Чтобы автоматически выделить найденный текст, добавьте его индекс во встроенный тег SEL:

```
where = self.text.search(target, INSERT, END)  # поиск от курсора вставки
pastit = where + ('+%dc' % len(target))        # индекс за найденной строкой
self.text.tag_add(SEL, where, pastit)          # пометить найденную строку тегом и выделить
self.text.focus()                              # выбрать сам элемент text
```

Если требуется выделить только одну строку, нужно сначала выполнить вызов tag_remove, о котором говорилось выше, – этот код добавляет выделение к тем, которые уже существуют (допускается создание на экране нескольких выделений). В целом можно добавлять в тег любое количество подстрок и обрабатывать их группой.

Подведем итоги: индексы, метки и адреса тегов можно использовать во всех случаях, когда требуется указать позицию в тексте. Например, метод see прокручивает изображение, чтобы сделать позицию видимой; он принимает все три типа задания позиции:

```
self.text.see('1.0')          # прокрутить экран наверх
self.text.see(INSERT)         # прокрутить экран до пометки курсора вставки
self.text.see(SEL_FIRST)      # прокрутить экран до тега выделения
```

Теги также могут применяться для форматирования и привязки событий, но эти детали будут отложены до конца раздела.

Операции редактирования текста

В примере 8.11 применены некоторые из этих понятий. В нем вводится поддержка четырех частых операций редактирования – сохранения в файле (*save*), удаления (*cut*) и вставки (*paste*) текста и поиск (*find*) строки – с помощью создания подкласса ScrolledText с дополнительными кнопками и методами. В графическом элементе Text есть набор готовых привязок клавиш, выполняющих некоторые частые операции редактирования, но они подражают редактору Unix Emacs и несколько непонятны; чаще в текстовом редакторе GUI предоставляются интерфейсы GUI операций редактирования, что более дружественно по отношению к пользователю.

Пример 8.11. PP2E\Gui\Tour\simpleedit.py

```
##############################################################################
# с помощью наследования добавить обычные средства редактирования в элемент
```

```
# с прокруткой текста; композиция (встраивание) тоже могла бы здесь действовать;
# ненадежно! надмножество функций есть в PyEdit;
##############################################################################
from Tkinter import *
from tkSimpleDialog import askstring
from tkFileDialog   import asksaveasfilename
from quitter        import Quitter
from scrolledtext   import ScrolledText             # наш, а не из стандартной поставки Python
class SimpleEditor(ScrolledText):                   # дополнительные функции есть в PyEdit
    def __init__(self, parent=None, file=None):
        frm = Frame(parent)
        frm.pack(fill=X)
        Button(frm, text='Save',  command=self.onSave).pack(side=LEFT)
        Button(frm, text='Cut',   command=self.onCut).pack(side=LEFT)
        Button(frm, text='Paste', command=self.onPaste).pack(side=LEFT)
        Button(frm, text='Find',  command=self.onFind).pack(side=LEFT)
        Quitter(frm).pack(side=LEFT)
        ScrolledText.__init__(self, parent, file=file)
        self.text.config(font=('courier', 9, 'normal'))
    def onSave(self):
        filename = asksaveasfilename()
        if filename:
            alltext = self.gettext()                        # с первого до последнего
            open(filename, 'w').write(alltext)              # сохранить текст в файле
    def onCut(self):
        text = self.text.get(SEL_FIRST, SEL_LAST)           # ошибка, если нет выделения
        self.text.delete(SEL_FIRST, SEL_LAST)               # следует обернуть в try
        self.clipboard_clear()
        self.clipboard_append(text)
    def onPaste(self):                                      # добавить текст из буфера
        try:
            text = self.selection_get(selection='CLIPBOARD')
            self.text.insert(INSERT, text)
        except TclError:
            pass                                            # не вставлять
    def onFind(self):
        target = askstring('SimpleEditor', 'Search String?')
        if target:
            where = self.text.search(target, INSERT, END)  # из курсора вставки
            if where:                                       # возвращает индекс
                print where
                pastit = where + ('+%dc' % len(target))     # индекс мимо цели
               #self.text.tag_remove(SEL, '1.0', END)      # снять выделение
                self.text.tag_add(SEL, where, pastit)       # выделить найденную цель
                self.text.mark_set(INSERT, pastit)          # установить пометку вставки
                self.text.see(INSERT)                       # прокрутить экран
                self.text.focus()                           # выбрать элемент text

if __name__ == '__main__':
    try:
        SimpleEditor(file=sys.argv[1]).mainloop()    # имя файла в командной строке
    except IndexError:
        SimpleEditor().mainloop()                    # или нет
```

Этот код также был написан с оглядкой на многократное использование – определяемый в нем класс `SimpleEditor` можно прикрепить в другом коде GUI или создать для него подкласс. Однако, как будет разъяснено в конце раздела, код не настолько надежен, как требуется от библиотечного инструмента общего назначения. Тем не менее

в нем реализован действующий текстовый редактор с помощью переносимого кода небольшого объема. При самостоятельном выполнении он выводит окно, представленное на рис. 8.17 (показано выполнение под Windows); после каждой успешной операции поиска в `stdout` выводятся позиции индексов:

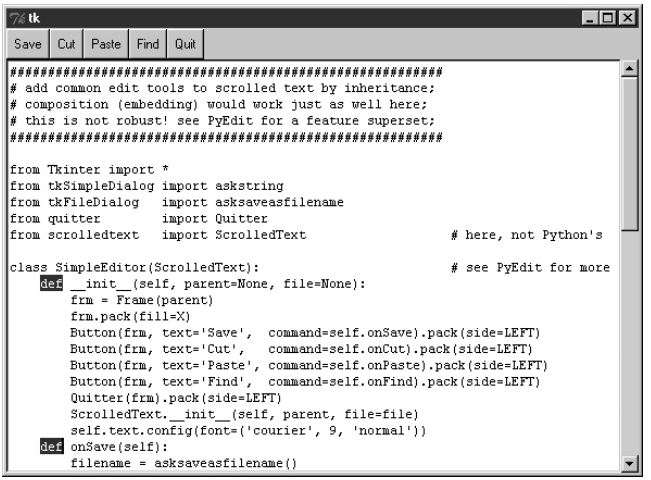


Рис. 8.17. simpleedit в действии

```
C:\...\PP2E\Gui\Tour>python simpleedit.py simpleedit.py
PP2E scrolledtext
14.4
24.4
```

Операция сохранения выводит имеющийся в Tkinter стандартный диалог сохранения, который выглядит естественным для каждой платформы образом. На рис. 8.18 показан этот диалог в Windows. Операция поиска тоже выводит стандартное диалоговое окно для ввода строки поиска (рис. 8.19); в полноценном редакторе можно было бы сохранить эту строку для повторного поиска (что мы и сделаем в PyEdit в следующей главе).

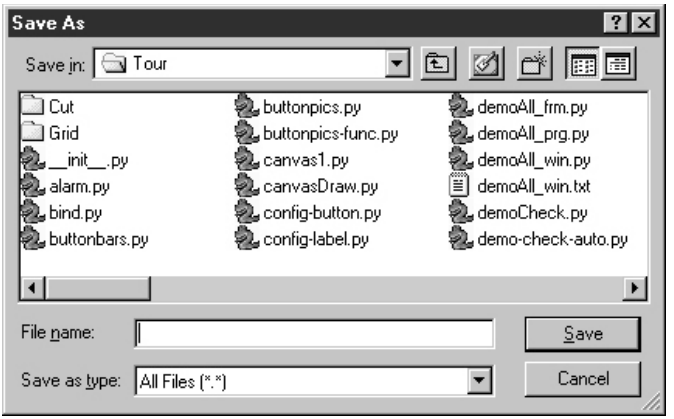


Рис. 8.18. Всплывающий диалог сохранения в Windows

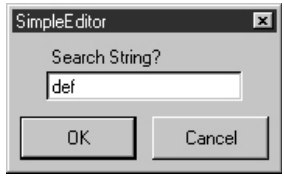


Рис. 8.19. Всплывающий диалог поиска

Использование буфера

Помимо операций с графическим элементом текста пример 8.18 применяет интерфейсы Tkinter к буферу обмена для реализации функций удаления и вставки. В совокупности эти операции позволяют перемещать текст в файле (вырезать в одном месте и вставить в другом). Используемый ими буфер обмена служит местом для временного хранения данных – удаленный текст помещается в буфер при вырезании и берется оттуда при вставке. Если ограничить поле зрения одной только этой программой, не будет причин, по которым строку текста нельзя было бы просто сохранить в переменной Python. Но буфер обмена является значительно более широким понятием.

Используемый в этом сценарии буфер является интерфейсом к *общесистемному* хранилищу, совместно используемому всеми программами компьютера. Поэтому его можно использовать для передачи данных между приложениями, в числе которых могут быть такие, которые понятия не имеют о Tkinter. Например, текст, вырезанный или скопированный в сеансе Microsoft Word, можно вставлять в окно `SimpleEditor`, а текст, вырезанный в `SimpleEditor`, можно вставить в Блокнот (можете попробовать). Используя для вырезания и вставки буфер обмена, `SimpleEditor` автоматически интегрируется с оконной системой в целом. Более того, буфер обмена используется не одним графическим элементом текста – его можно применять для вырезания и вставки графических объектов в графическом элементе `Canvas` (рассматриваемом следующим).

Использованный в данном сценарии базовый интерфейс буфера Tkinter выглядит так:

```
self.clipboard_clear()                          # очистить буфер
self.clipboard_append(text)                     # сохранить в нем строку текста
text = self.selection_get(selection='CLIPBOARD') # получить содержимое, если оно есть
```

Все эти вызовы доступны в виде методов, наследуемых всеми объектами графических элементов Tkinter, поскольку они являются глобальными по своей природе. Использованное в этом сценарии выделение `CLIPBOARD` применимо на всех платформах (существует еще выделение `PRIMARY`, но обычно им можно пользоваться только в X Windows, поэтому мы здесь его не рассматриваем). Обратите внимание, что вызов `selection_get` возбуждает в случае неудачи исключительную ситуацию `TclError`; данный сценарий ее просто игнорирует и прекращает запрос вставки, но в дальнейшем мы будем поступать лучше.

Композиция в сравнении с наследованием

В данном примере `SimpleEditor` расширяет `ScrolledText` дополнительными кнопками и методами обратных вызовов с помощью *наследования*. Как было показано, допускается прикреплять (встраивать) объекты GUI, написанные как компоненты, подобно `ScrolledText`. Модель, в которой компоненты прикрепляются, обычно называется *композицией*; некоторые считают, что она более понятна и реже приводит к конфликту имен, чем расширение путем наследования.

Чтобы дать представление о различиях между этими двумя подходами, ниже приведен набросок кода, который следовало бы написать, чтобы прикрепить ScrolledText к SimpleEditor. Измененные строки выделены в нем полужирным шрифтом (полную реализацию композиции можно найти на CD в файле *simpleedit-2.py*). В основном задача состоит в указании правильных родительских элементов и добавлении атрибута «st» для получения методов графического элемента Text:

```
class SimpleEditor(Frame):
    def __init__(self, parent=None, file=None):
        Frame.__init__(self, parent)
        self.pack()
        frm = Frame(self)
        frm.pack(fill=X)
        Button(frm, text='Save',  command=self.onSave).pack(side=LEFT)
        ...еще строки...
        Quitter(frm).pack(side=LEFT)
        self.st = ScrolledText(self, file=file)           # прикрепление вместо подкласса
        self.st.text.config(font=('courier', 9, 'normal'))
    def onSave(self):
        filename = asksaveasfilename()
        if filename:
            alltext = self.st.gettext()                   # пройти через атрибут
            open(filename, 'w').write(alltext)
    def onCut(self):
        text = self.st.text.get(SEL_FIRST, SEL_LAST)
        self.st.text.delete(SEL_FIRST, SEL_LAST)
        ...еще строки...
```

При выполнении такого кода получается схожее окно. Судите сами, что здесь лучше – композиция или наследование. Если правильно кодировать классы Python GUI, они будут работать в любом режиме.

«Простым» он назван обоснованно

Наконец, пока вы еще не записали в системном реестре, что по умолчанию средством просмотра текстовых файлов служит SimpleEditor («простой редактор»), я должен заметить, что хотя он обладает всеми основными функциями, но является в некотором роде урезанной версией редактора PyEdit, с которым мы познакомимся в главе 9. На самом деле вам нужно сейчас же посмотреть соответствующий пример, если вы ищете более полный код Tkinter для обработки текста. Графический элемент text обладает такой мощью, что трудно продемонстрировать какие-то другие его возможности в коде меньшего размера, чем тот, который уже есть в программе PyEdit.

Я должен также отметить, что SimpleEditor не только ограничен по функциям, но и весьма небрежен – многие граничные случаи проходят без проверки и вызывают неперехватываемые исключительные ситуации, которые не убивают GUI, но и не обрабатываются и не выводят сообщений. Даже о перехваченных ошибках пользователю не сообщается (например, о вставках, в которых нечего вставлять). Обязательно посмотрите PyEdit как пример более устойчивой и полной реализации операций, введенных в SimpleEditor.

Более сложные операции с тегами в Text

Помимо указания позиции, теги текста могут применяться для форматирования и действий как над всеми символами подстроки, так и над всеми подстроками, поме-

щенными в тег. В действительности это составляет значительную часть мощи, предоставляемой графическим элементом текста:

- У тегов есть атрибуты *форматирования* для установки цвета, шрифта, табуляции, межстрочных интервалов и выравнивания; они могут применяться одновременно для нескольких частей текста, если связать их с тегом и выполнить форматирование тега с помощью метода `tag_config`, во многом подобного используемому нами общему методу `config` графических элементов.
- С тегами можно ассоциировать *привязку событий*, которая позволяет реализовывать такие вещи, как организация гиперссылок в графическом элементе `Text`: щелчок по тексту вызывает обработчик события его тега. Привязка тегов организуется с помощью метода `tag_bind`, во многом подобного уже знакомому общему методу `bind` графических элементов.

С помощью тегов можно выводить различные настройки внутри одного и того же графического элемента текста; например можно применить некоторый шрифт к элементу текста в целом и другие шрифты к тексту в тегах. Кроме того, элемент текста позволяет встраивать другие графические элементы по индексу (они воспринимаются как отдельный символ), а также графические изображения.

Пример 8.12 иллюстрирует основы сразу всех этих развитых средств и изображает интерфейс, показанный на рис. 8.20. В этом сценарии форматирование и привязка событий применяются к трем подстрокам, помеченным тегами, текст выводится с помощью двух разных схем шрифтов и цветов, а также встраиваются графическое изображение и кнопка. Двойной щелчок мышью по любой из подстрок, заключенных в теги (или по встроенной кнопке), генерирует событие, которое выводит в `stdout` сообщение «Got tag event».

Пример 8.12. PP2E\Gui\Tour\texttags.py

```
# демонстрация усложненных интерфейсов тегов и текста
from Tkinter import *
root = Tk()
def hello(event): print 'Got tag event'
# создать и настроить Text
text = Text()
text.config(font=('courier', 15, 'normal'))                  # установить шрифт для всех
символов
text.config(width=20, height=12)
text.pack(expand=YES, fill=BOTH)
text.insert(END, 'This is\n\nthe meaning\n\nof life.\n\n')   # вставить 6 строчек
# встроить окна и изображения
btn = Button(text, text='Spam', command=lambda: hello(0))    # встроить кнопку
btn.pack()
text.window_create(END, window=btn)                          # встроить изображение
text.insert(END, '\n\n')
img = PhotoImage(file='../gifs/PythonPowered.gif')
text.image_create(END, image=img)
# применить теги к подстрокам
text.tag_add('demo', '1.5', '1.7')                           # тег 'is'
text.tag_add('demo', '3.0', '3.3')                           # тег 'the'
text.tag_add('demo', '5.3', '5.7')                           # тег 'life'
text.tag_config('demo', background='purple')                 # изменить цвета в теге
text.tag_config('demo', foreground='white')                  # здесь не вызываются bg/fg
text.tag_config('demo', font=('times', 16, 'underline'))     # изменить шрифт в теге
text.tag_bind('demo', '<Double-1>', hello)                   # привязать события в теге
root.mainloop()
```

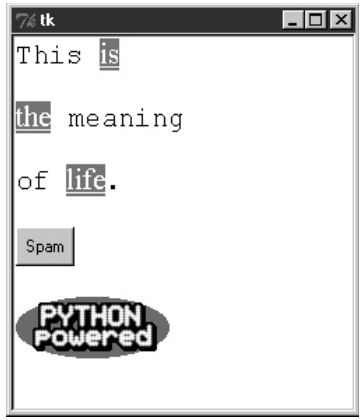


Рис. 8.20. Теги Text в действии

Такие средства встраивания и тегов можно в конечном итоге использовать для вывода веб-страницы. А стандартный модуль анализатора HTML `htmllib` может помочь с автоматизацией построения GUI веб-страницы. Как можно догадаться, графический элемент текста предоставляет больше возможностей программирования GUI, чем позволяет описать объем книги. За подробностями о возможностях, предоставляемых тегами и текстом, следует обратиться к другим справочникам по Tk и Tkinter. А сейчас начнутся занятия в художественной школе.

Графический элемент Canvas

Что касается графики, то графический элемент Tkinter `Canvas` (холст) является самым свободным по форме инструментом в этой библиотеке. Здесь можно рисовать фигуры, динамически перемещать объекты и располагать другие типы графических элементов. Холст основан на структурированной модели графического объекта: все, что изображается на холсте, может обрабатываться как объект. Можно спуститься до уровня обработки пикселов, а можно оперировать более крупными объектами, такими как фигуры, изображения и встроенные графические элементы.

Базовые операции с Canvas

Холст повсеместно используется во многих нетривиальных GUI, и далее в этой книге можно будет увидеть более крупные примеры холстов с именами PyDraw, PyView, PyClock и PyTree. А сейчас сразу займемся примером, в котором демонстрируются основы его применения. В примере 8.13 выполняется большинство главных методов создания изображений на холсте.

Пример 8.13. PP2E\Gui\Tour\canvas1.py

```
# демонстрация всех главных интерфейсов canvas
from Tkinter import *
canvas = Canvas(width=300, height=300, bg='white')   # 0,0 служит левым верхним углом
canvas.pack(expand=YES, fill=BOTH)                   # рост вниз, вправо
canvas.create_line(100, 100, 200, 200)               # fromX, fromY, toX, toY
canvas.create_line(100, 200, 200, 300)               # нарисовать фигуры
for i in range(1, 20, 2):
    canvas.create_line(0, i, 50, i)
```

```
canvas.create_oval(10, 10, 200, 200, width=2, fill='blue')
canvas.create_arc(200, 200, 300, 100)
canvas.create_rectangle(200, 200, 300, 300, width=5, fill='red')
canvas.create_line(0, 300, 150, 150, width=10, fill='green')
photo=PhotoImage(file='../gifs/guido.gif')
canvas.create_image(250, 0, image=photo, anchor=NW)  # встроить фотографию
widget = Label(canvas, text='Spam', fg='white', bg='black')
widget.pack()
canvas.create_window(100, 100, window=widget)        # встроить графический элемент
canvas.create_text(100, 280, text='Ham')             # нарисовать текст
mainloop()
```

При выполнении этого сценария выводится окно, показанное на рис. 8.21. Ранее уже было показано, как поместить на холст графическое изображение и установить соответствующие ему размеры холста (см. раздел «Изображения» в конце главы 7). Этот сценарий изображает также фигуры, текст и даже встроенный графический элемент `Label`. Его окно ограничивается лишь изображением; несколько далее будет показано, как добавить обратные вызовы событий, дающие пользователю возможность взаимодействовать с отображаемыми элементами.

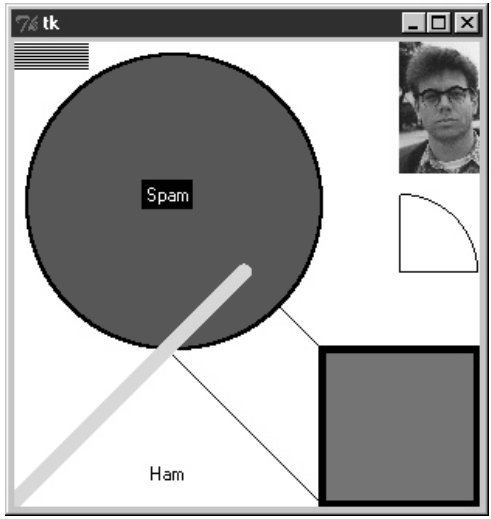


Рис. 8.21. Жестко зашитые в canvas1 изображения объектов

Программирование графического элемента Canvas

Пользоваться холстами легко, но они используют систему координат, определяют уникальные методы отображения и именуют объекты с помощью идентификаторов или тегов. Данный раздел знакомит с этими базовыми понятиями холста.

Координаты

Все предметы, отображаемые на холсте, представляют собой отдельные объекты, но в действительности не являются графическими элементами (widgets). Если внимательно изучить сценарий `canvas1`, то можно заметить, что холсты создаются и *упаковываются* (или размещаются по сетке или располагаются) в родительском контейнере так же, как любые другие графические элементы Tkinter. Однако к предметам, изобра-

жаемым на холсте, это не относится: фигуры, образы и т. п. располагаются и перемещаются по холсту с помощью координат, идентификаторов и тегов. При этом координаты составляют наиболее фундаментальную часть модели холста.

Холст определяет в качестве области отображения систему координат (X,Y); X обозначает горизонтальную ось, Y – вертикальную. По умолчанию координаты измеряются в пикселах (точках); левый верхний угол холста имеет координаты (0,0), а координаты X и Y возрастают вправо и вниз, соответственно. Чтобы рисовать и встраивать объекты на холсте, задаются одна или более пар координат (X,Y), указывающие абсолютные местоположения на холсте. Такой способ отличается от ограничений, использовавшихся до сих пор для упаковки графических элементов, но позволяет управлять графической структурой с очень большой точностью и поддерживает более свободные по форме технологии интерфейсов, например анимацию.[1]

Создание объекта

Холст позволяет рисовать и отображать обычно встречающиеся *фигуры* типа линий, овалов, прямоугольников, дуг и многоугольников. Кроме того, можно встраивать текст, образы и другие типы графических элементов Tkinter, например метки и кнопки. В сценарии `canvas1` продемонстрированы все основные вызовы конструкторов графических объектов; каждому из них передается один или более наборов координат (X,Y), задающих местоположение нового объекта, начальные и конечные точки или противоположные углы рамки, содержащей фигуру:

```
id = canvas.create_line(fromX, fromY, toX, toY)       # начало, конец отрезка прямой
id = canvas.create_oval(fromX, fromY, toX, toY)       # два противоположных угла овала
id = canvas.create_arc( fromX, fromY, toX, toY)       # два противоположных угла дуги
id = canvas.create_rectangle(fromX, fromY, toX, toY)  # два противоположных угла прямоугольника
```

В других вызовах для вычерчивания указывается только одна пара (X,Y), задающая местоположение левого верхнего угла объекта:

```
id = canvas.create_image(250, 0, image=photo, anchor=NW)  # встроить фотографию
id = canvas.create_window(100, 100, window=widget)        # встроить графический элемент
id = canvas.create_text(100, 280, text='Ham')             # нарисовать некоторый текст
```

Холст также предоставляет метод `create_polygon`, который принимает произвольное множество аргументов координат, определяющих конечные точки соединенных вместе линий; он удобен для вычерчивания весьма произвольных фигур, образованных из прямых линий.

Помимо координат большинство вызовов для черчения позволяет задавать обычные для всех параметры настройки, такие как ширина границы, цвет заливки и цвет границы и т. д. У некоторых типов объектов есть собственные уникальные параметры настройки; например, для линий может быть указана форма необязательной стрелки, а текст, графические элементы и изображения можно привязывать по направлениям сторон света (что похоже на `anchor` упаковщика, но в действительности задает точку объекта, помещаемую в указанные в вызове координаты (X,Y); `NW`, например, помещает в (X,Y) левый верхний угол объекта).

---

[1] О технике анимации рассказывается в конце данного обзора. Поскольку в область отрисовки холста можно встраивать другие графические элементы, координатная система делает их идеальными для реализации GUI, дающих пользователям возможность создания других GUI путем перетаскивания встроенных графических элементов по холсту – полезное применение холста, которое мы изучили бы в этой книге, будь у меня несколько сотен лишних страниц.

Важнее всего, вероятно, отметить здесь, что Tkinter по большей части сам выполняет «черновую» работу – рисуя что-нибудь, вы указываете координаты, и фигуры автоматически вычерчиваются и отображаются посредством пикселов. Если вам когда-либо приходилось заниматься графикой на низком уровне, то вы оцените разницу.

Идентификаторы объектов и операции

Сценарий `canvas1` не использует той особенности, что у каждого помещаемого на холст объекта есть идентификатор. Его возвращает метод `create_`, который вычерчивает или встраивает объект (он кодировался как «id» в примерах предыдущего раздела). Этот идентификатор можно впоследствии передавать другим методам, перемещающим объект в новые координаты, устанавливающим параметры его настройки, удаляющим его с холста, поднимающим или опускающим его относительно других перекрывающихся объектов и т. д.

Например, метод холста `move` принимает как идентификатор объекта, так и смещения (не координаты) X и Y и передвигает объект согласно заданному смещению:

```
canvas.move(objectIdOrTag, offsetX, offsetY)    # переместить объект(ы) согласно смещению
```

Если при этом объект передвигается за пределы экрана, он просто обрезается (не показывается). Объекты могут также обрабатываться другими часто встречающимися операциями:

```
canvas.delete(objectIdOrTag)                    # удалить объект(ы) с холста
canvas.tkraise(objectIdOrTag)                   # поднять объект(ы) вверх
canvas.lower(objectIdOrTag)                     # опустить объект(ы) вниз
canvas.itemconfig(objectIdOrTag, fill='red')    # залить объект(ы) красным цветом
```

Обратите внимание на имя `tkraise` – слово `raise` является в Python зарезервированным. Заметьте также, что для настройки объектов, изображенных на холсте, после их создания используется метод `itemconfig`; метод `config` применяется для изменения параметров самого холста. Однако главное, что нужно отметить, это возможность обработать сразу весь графический объект, поскольку Tkinter основывается на структурированных объектах; не нужно поднимать и перерисовывать каждый пиксел вручную, чтобы осуществить перемещение или подъем объекта.

Теги объекта Canvas

Однако дело обстоит еще лучше: помимо идентификаторов объектов существуют теги. Тег – создаваемое программистом имя, с которым можно связать ряд отображаемых объектов и применить операцию холста сразу ко всей группе. Пометка объектов тегами в `Canvas` по крайней мере по духу сходна с пометкой тегами подстрок в элементе `Text`, изученной в предшествующем разделе. В общих чертах методы холста принимают идентификатор отдельного объекта или имя тега.

Например, можно переместить целую группу отображаемых объектов, привязав их к одному и тому же тегу и передав его имя методу `move` холста. В действительности это та причина, по которой `move` принимает смещения, а не координаты – при задании тега каждый ассоциированный с ним объект перемещается соответственно одинаковым смещениям (X,Y); при задании абсолютных координат все связанные с тегом объекты оказались бы в одном и том же месте друг над другом.

Чтобы связать объект с тегом, нужно задать имя тега в параметре `tag` вызова, отображающего объект, или вызвать метод холста `addtag_withtag(tag, objectIdOrTag)` или родственный ему. Например:

```
canvas.create_oval(x1, y1, x2, y2, fill='red', tag='bubbles')
canvas.create_oval(x3, y3, x4, y4, fill='red', tag='bubbles')
```

```
objectId = canvas.create_oval(x5, y5, x6, y6, fill='red')
canvas.addtag_withtag('bubbles', objectId)
canvas.move('bubbles', diffx, diffy)
```

Этот код создает три овала и перемещает их одновременно благодаря связыванию с одним и тем же именем тега. У многих объектов может быть один и тот же тег, многие теги могут ссылаться на один и тот же объект, и каждый тег можно конфигурировать и обрабатывать независимо.

Как и в `Text`, для графических элементов `Canvas` есть предопределенные имена: тег «all» ссылается на все объекты, имеющиеся на холсте, а «current» ссылается на тот объект, который находится под курсором мыши. Можно не только запрашивать объект под мышью, но и осуществлять *поиск* объектов с помощью методов `find_` холста: `canvas.find_closest(X,Y)`, к примеру, возвращает набор, первый элемент которого содержит идентификатор объекта, находящегося ближе всего к точке с заданными координатами, – это удобно, когда уже есть координаты, полученные в обратном вызове общего события, сгенерированного щелчком мыши.

К представлению о тегах холста мы вернемся снова в более позднем примере из этой главы (если вам не терпится, можете посмотреть сценарии анимации ближе к концу). Холсты поддерживают другие операции и параметры, для рассказа о которых здесь недостаточно места (например, метод холста `postscript` позволяет сохранить холст в файле postscript). Дополнительные сведения можно найти в примерах, имеющихся далее в книге, таких как PyDraw, а в справочниках по Tk или Tkinter есть полный список параметров объекта холст.

Прокрутка холстов

Как демонстрирует пример 8.14, полосы прокрутки можно перекрестно связывать с холстами с помощью тех же протоколов, которые ранее использовались для добавления их в окна списков и текст, но с некоторыми особыми требованиями.

Пример 8.14. PP2E\Gui\Tour\scrolledcanvas.py

```
from Tkinter import *
class ScrolledCanvas(Frame):
    def __init__(self, parent=None, color='brown'):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)                  # сделать расширяемым
        canv = Canvas(self, bg=color, relief=SUNKEN)
        canv.config(width=300, height=200)                # размер области вывода
        canv.config(scrollregion=(0,0,300, 1000))         # углы холста
        canv.config(highlightthickness=0)                 # без границы

        sbar = Scrollbar(self)
        sbar.config(command=canv.yview)                 # связать sbar и canv
        canv.config(yscrollcommand=sbar.set)            # перемещение одного перемещает другой
        sbar.pack(side=RIGHT, fill=Y)                   # упакован первым-обрезается последним
        canv.pack(side=LEFT, expand=YES, fill=BOTH)     # canv обрезается первым

        for i in range(10):
            canv.create_text(150, 50+(i*100), text='spam'+str(i), fill='beige')
        canv.bind('<Double-1>', self.onDoubleClick)       # установить обработчик события
        self.canvas = canv
    def onDoubleClick(self, event):
        print event.x, event.y
        print self.canvas.canvasx(event.x), self.canvas.canvasy(event.y)
if __name__ == '__main__': ScrolledCanvas().mainloop()
```

Этот сценарий создает окно, показанное на рис. 8.22. Оно аналогично предыдущим примерам прокрутки, но в прокрутке холстов есть две особенности:

- Можно задать размер окна, в которое производится вывод, и обязательно задается размер прокручиваемого холста в целом.
- Кроме того, нужно устанавливать соответствие между координатами области просмотра и координатами всего холста, если холст больше, чем область его просмотра. При прокрутке холст почти всегда больше, чем его видимая часть, поэтому часто требуется отображать координаты.

Рис. 8.22. scrolledcanvas в действии

Размеры задаются как параметры настройки. Размер *области просмотра* задается с помощью параметров холста `width` и `height`. Чтобы задать общий размер *холста*, в наборе, передаваемом параметром `scrollregion`, указываются координаты (X,Y) левого верхнего и правого нижнего углов холста. Если размер области просмотра не задан, используется размер по умолчанию. Если не задан `scrollregion`, он принимается по умолчанию равным размеру области просмотра; при этом полоса прокрутки становится бесполезной, так как область просмотра оказывается содержащей весь холст целиком.

Отображение координат осуществляется несколько более сложным образом. Если прокручиваемая область просмотра для холста оказывается меньше, чем холст в целом, то координаты (X,Y), возвращаемые объектами событий, представляют собой координаты в области просмотра, а не в холсте в целом. Обычно требуется перевести координаты события в координаты холста, для чего они передаются методам `canvasx` и `canvasy` прежде, чем быть примененными для обработки объектов.

Например, если запустить сценарий прокручиваемого холста и посмотреть на сообщения, выводимые при двойных щелчках мышью, можно заметить, что координаты события всегда относятся к окну просмотра, а не холсту в целом:

```
C:\...\PP2E\Gui\Tour>python scrolledcanvas.py
2 0                     x,y события, если холст прокручен доверху
2.0 0.0                 x,y холста – те же, если нет пикселов границы
150 106
150.0 106.0
299 197
299.0 197.0
3 2                     x,y события, если холст прокручен донизу
3.0 802.0               x,y холста – у совсем другой
296 192
296.0 992.0
152 97                  при прокрутке в среднюю часть холста
```

```
152.0 599.0
16 187
16.0 689.0
```

Здесь отображаемый X холста всегда совпадает с X холста, поскольку область отображения и холст имеют одинаковую ширину 300 пикселов (расхождение могло быть в два пиксела благодаря автоматической установке границ, если бы не значение `highlightthickness`, установленное в сценарии). Но обратите внимание, что отображаемый Y становится совершенно отличным от Y холста, если щелкнуть по вертикальной полосе прокрутки. Без преобразования координат значение Y события неверно указывало бы на место, находящееся на холсте значительно выше.

Для большинства примеров в этой книге, использующих холсты, такого преобразования не требуется – (0,0) всегда соответствует левому верхнему углу изображения холста, в котором происходит щелчок мышью – но лишь потому, что холсты не прокручиваются. Однако посмотрите на пример с программой PyTree, приведенный далее в этой книге и демонстрирующий холст с обеими полосами прокрутки – горизонтальной и вертикальной – и динамически изменяемым размером области прокрутки.

На практике, если холст прокручивается, необходимо в обработчиках обратного вызова, учитывающих позицию, преобразовывать координаты события в действительные координаты холста. Для некоторых обработчиков, привязанных к отдельным изображенным объектам, а не ко всему холсту, это безразлично; причина станет ясна, когда мы более подробно поговорим о событиях.

События холстов

Как и в случае `Text` и `Listbox`, для `Canvas` нет понятия одного обратного вызова, задаваемого параметром `command`. Вместо этого программы, содержащие холсты, обычно используют другие графические элементы или вызов низкого уровня `bind`, чтобы установить обработчики для щелчков мыши, нажатия клавиш и т. п., как в примере 8.14. Пример 8.15 показывает, как привязать события самого холста, чтобы реализовать некоторые наиболее частые операции изображения на холсте.

Пример 8.15. PP2E\Gui\Tour\canvasDraw.py

```
##############################################################
# вычерчивание на холсте гибких фигур при перемещении мыши с нажатой правой
# кнопкой; расширения с тегами и анимацией см. в canvasDraw_tags*.py
##############################################################

from Tkinter import *
trace = 0

class CanvasEventsDemo:
    def __init__(self, parent=None):
        canvas = Canvas(width=300, height=300, bg='beige')
        canvas.pack()
        canvas.bind('<ButtonPress-1>', self.onStart)      # щелкнуть
        canvas.bind('<B1-Motion>',     self.onGrow)       # и тащить
        canvas.bind('<Double-1>',      self.onClear)      # удалить все
        canvas.bind('<ButtonPress-3>', self.onMove)       # двигать последнюю
        self.canvas = canvas
        self.drawn  = None
        self.kinds = [canvas.create_oval, canvas.create_rectangle]
    def onStart(self, event):
        self.shape = self.kinds[0]
        self.kinds = self.kinds[1:] + self.kinds[:1]      # начать вытягивание
```

```
        self.start = event
        self.drawn = None
    def onGrow(self, event):                            # удалить и перерисовать
        canvas = event.widget
        if self.drawn: canvas.delete(self.drawn)
        objectId = self.shape(self.start.x, self.start.y, event.x, event.y)
        if trace: print objectId
        self.drawn = objectId
    def onClear(self, event):
        event.widget.delete('all')                      # тег all
    def onMove(self, event):
        if self.drawn:                                  # передвинуть в точку щелчка
            if trace: print self.drawn
            canvas = event.widget
            diffX, diffY = (event.x - self.start.x), (event.y - self.start.y)
            canvas.move(self.drawn, diffX, diffY)
            self.start = event

if __name__ == '__main__':
    CanvasEventsDemo()
    mainloop()
```

Этот сценарий перехватывает и обрабатывает три действия, производимые мышью:

Очистка холста

Чтобы удалить все имеющееся на холсте, сценарий привязывает событие двойного щелчка левой кнопкой к выполнению метода холста `delete` с тегом «all» – встроенным тегом, ассоциированным с каждым объектом на экране. Обратите внимание, что доступ к графическому объекту холста, по которому сделан щелчок, осуществляется через объект события, передаваемый обработчику обратного вызова (он также доступен через `self.canvas`).

Вытягивание фигур

При нажатии левой кнопки мыши и перетаскивании (перемещении при нажатой кнопке) создается прямоугольник или овал. При этом фигура увеличивается и сжимается, как резиновая, и окончательный ее размер и положение определяются тем, в какой точке отпущена кнопка мыши.

Чтобы выполнить такую работу с помощью Tkinter, нужно лишь при каждой генерации события перетаскивания стереть старую фигуру и нарисовать новую; обе операции удаления и вычерчивания осуществляются настолько быстро, что возникает эффект эластичного вытягивания. Конечно, чтобы начертить фигуру в соответствии с текущим положением мыши, нужна начальная точка, а чтобы удалить фигуру перед тем, как вычерчивать новую, нужно запоминать идентификатор объекта, нарисованного последним. Здесь участвуют два события: исходное событие *нажатия кнопки* сохраняет начальные координаты (точнее, объект события нажатия кнопки, который содержит начальные координаты), а события *перемещения мыши* стирают прежнюю фигуру и рисуют новую от начальных координат до новых координат мыши, сохраняя ID нового объекта для операции стирания в следующем событии.

Перемещение объекта

При щелчке правой кнопкой мыши (кнопкой 3) сценарий сразу перемещает объект, нарисованный последним, в то место, где произведен щелчок. Аргумент события дает координаты (X,Y) щелчка, из которых вычитаются начальные координаты последнего нарисованного объекта, чтобы получить *смещения* (X,Y), передаваемые методу холста move (вспомним, что move не должен получать координаты).

Не следует забывать о необходимости сначала преобразовать координаты события, если холст прокручен.

В итоге после взаимодействия с пользователем получается окно, подобное изображенному на рис. 8.23. При вытягивании объектов сценарий поочередно вычерчивает овалы и прямоугольники. Установите в сценарии глобальную переменную `trace` и вы увидите выводимые в `stdout` идентификаторы новых объектов, вычерчиваемых при перетаскивании. Этот снимок экрана получен после нескольких вытягиваний и перемещений объектов, чего не скажешь, глядя на него. Запустите пример на своем компьютере, чтобы лучше ощутить выполняемые им операции.

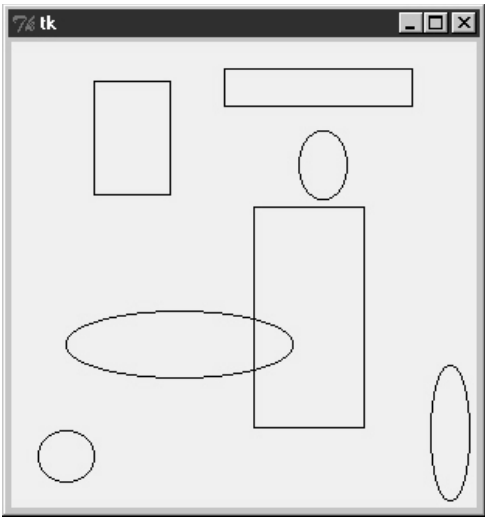


Рис. 8.23. canvasDraw после нескольких вытягиваний и перемещений

Привязка событий к конкретным элементам

Во многом подобно тому, как это делалось для графического элемента `Text`, возможна привязка событий к одному или нескольким конкретным объектам, нарисованным на `Canvas` с помощью его метода `tag_bind`. Этот метод принимает в качестве первого аргумента строку с именем тега или ID объекта. Например, можно зарегистрировать отдельные обработчики событий щелчков мыши для каждого нарисованного элемента или для любого из группы нарисованных и помеченных тегом элементов вместо обработчика для холста в целом. В примере 8.16 для иллюстрации интерфейсов обработчик двойного щелчка привязывается как к самому холсту, так и к двум конкретным текстовым элементам на нем; при его выполнении создается окно, показанное на рис. 8.24.

Пример 8.16. PP2E\Gui\Tour\canvas-bind.py

```
from Tkinter import *

def onCanvasClick(event):
    print 'Got canvas click', event.x, event.y, event.widget
def onObjectClick(event):
    print 'Got object click', event.x, event.y, event.widget,
    print event.widget.find_closest(event.x, event.y)   # найти id текстового объекта

root = Tk()
canv = Canvas(root, width=100, height=100)
```

```
obj1 = canv.create_text(50, 30, text='Click me one')
obj2 = canv.create_text(50, 70, text='Click me two')

canv.bind('<Double-1>', onCanvasClick)                  # привязать к холсту в целом
canv.tag_bind(obj1, '<Double-1>', onObjectClick)        # привязать к нарисованному элементу
canv.tag_bind(obj2, '<Double-1>', onObjectClick)        # теги тоже могут использоваться
canv.pack()
root.mainloop()
```

Рис. 8.24. Окно canvas-bind

Здесь в `tag_bind` передаются ID объектов, но строка с именем тега тоже действует. При щелчке в окне этого сценария снаружи текстовых элементов срабатывает обработчик события окна; при щелчке по любому текстовому элементу срабатывают оба обработчика – холста и элемента. Вот что будет выведено в `stdout` после двух щелчков по холсту и одного щелчка по каждому из текстовых элементов; для получения ID объекта для конкретного текстового элемента, ближайшего к точке щелчка, используется метод холста `find_closest`:

```
C:\...\PP2E\Gui\Tour>python canvas-bind.py
Got canvas click 3 6 .8217952                 щелчки по холсту
Got canvas click 46 52 .8217952
Got object click 51 33 .8217952 (1,)          щелчок по первому тексту
Got canvas click 51 33 .8217952
Got object click 55 69 .8217952 (2,)          щелчок по второму тексту
Got canvas click 55 69 .8217952
```

Мы еще раз вернемся к идее событий, привязываемых к холсту, в примере PyDraw в главе 9, где они будут использованы для реализации богатой функциями программы рисования и перемещения. Мы также вернемся к сценарию `canvasDraw` далее в этой главе, где мы добавим перемещения, основанные на тегах и простую анимацию с помощью инструментов, использующих время, поэтому сделайте закладку на этой странице для справок в дальнейшем. Однако сначала давайте свернем немного в сторону и исследуем другой способ расположения графических элементов в окнах.

Сетки

До сих пор мы размещали на экране графические элементы, вызывая их метод `pack` – интерфейс к упаковывающему менеджеру геометрии в Tkinter. В данном разделе мы познакомимся с `grid`, наиболее часто используемой альтернативой упаковщика.

Как уже говорилось, действие администраторов геометрии Tkinter заключается в размещении дочерних графических элементов внутри контейнера – родительского графического элемента (обычно родительскими являются элементы `Frame` или окна верхнего уровня). Когда графическому элементу предлагается упаковать себя или разместить на сетке, в действительности это просьба к его родительскому элементу расположить его среди собратьев. При использовании `pack` задаются ограничения, и менеджеру геометрии позволяется соответствующим образом расположить графичес-

кие элементы. При использовании grid графические элементы располагаются в родительском элементе-контейнере по рядам и колонкам, как если бы он был таблицей.

Расположение по сетке является в Tkinter совершенно отдельной системой управления геометрией. Действительно, на момент написания этой книги pack и grid взаимно исключают друг друга при расположении графических элементов в одном и том же окне – в одном контейнере можно упаковывать элементы или располагать по сетке, но не то и другое одновременно. Это разумно, если понять, что менеджеры геометрии выполняют свою работу в родительских элементах, и графический элемент может размещаться только одним менеджером геометрии.

Однако это означает, что по крайней мере внутри одного контейнера нужно выбрать grid или pack и придерживаться этого метода. Зачем тогда нужна сетка? В целом grid удобно использовать при создании экранов типа форм; расположение полей ввода по рядам и колонкам ничуть не сложнее, чем организация структуры экрана с помощью вложенных фреймов. Однако, как мы увидим, на практике grid не позволяет заметно уменьшить объем кода или сложность в сравнении с эквивалентными решениями, получаемыми путем упаковки, особенно если в GUI должны решаться задачи изменения размеров. Иными словами, выбор между двумя схемами размещения является в значительной мере вопросом стиля, а не технологии.

Основы работы с сеткой

Начнем с базовых понятий; в примере 8.17 размещается таблица из меток и полей ввода – уже знакомых нам графических элементов Label и Entry. В данном случае, однако, они располагаются на сетке.

Пример 8.17. PP2E\Gui\Tour\Grid\grid1.py

```
from Tkinter import *
colors = ['red', 'green', 'orange', 'white', 'yellow', 'blue']
r = 0
for c in colors:
    Label(text=c, relief=RIDGE,  width=25).grid(row=r, column=0)
    Entry(bg=c,   relief=SUNKEN, width=50).grid(row=r, column=1)
    r = r+1
mainloop()
```

При выполнении этого сценария создается окно, показанное на рис. 8.25, с данными, введенными в некоторые поля. И снова эта книга нехорошо обходится с цветами, отображаемыми в правой части, поэтому вам придется немного напрячь свое воображение (или запустить сценарий на своем компьютере).

Это классическая структура формы для ввода: метки в левой части описывают данные, которые должны быть введены в поля в правой части. Исключительно для развлечения этот сценарий выводит слева названия цветов, в которые окрашены соответ-

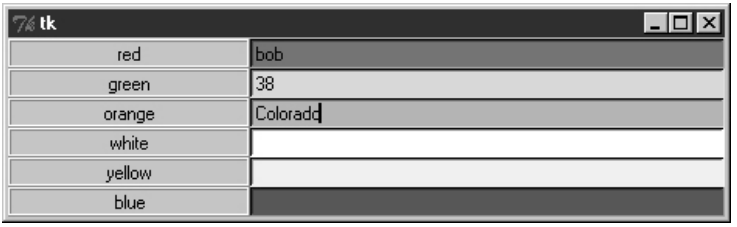


Рис. 8.25. Менеджер геометрии grid в псевдоживых цветах

ствующие поля ввода справа. Прелестный табличный вид достигается с помощью следующих двух строк:

```
Label(...).grid(row=r, column=0)
Entry(...).grid(row=r, column=1)
```

С точки зрения окна контейнера метка помещается в колонку 0 строки с текущим номером (счетчик начинается с 0), а поле ввода помещается в колонку 1. В итоге система размещения по сетке автоматически располагает метки и поля ввода в двумерной таблице, обеспечивая одинаковую ширину колонок, достаточную для размещения самого большого элемента в каждой колонке.

grid в сравнении с pack

Настало время сделать некоторые сравнения и противопоставления: пример 8.18 реализует одинаковые расцвеченные формы ввода с помощью как `grid`, так и `pack`, чтобы легче было обнаружить различия между двумя подходами.

Пример 8.18. PP2E\Gui\Tour\Grid\grid2.py

```
# добавить эквивалентное окно, использующее pack

from Tkinter import *
colors = ['red', 'green', 'yellow', 'orange', 'blue', 'navy']

def gridbox(parent):
    r = 0
    for c in colors:
        l = Label(parent, text=c, relief=RIDGE,  width=25)
        e = Entry(parent, bg=c,   relief=SUNKEN, width=50)
        l.grid(row=r, column=0)
        e.grid(row=r, column=1)
        r = r+1

def packbox(parent):
    for c in colors:
        f = Frame(parent)
        l = Label(f, text=c, relief=RIDGE,  width=25)
        e = Entry(f, bg=c,   relief=SUNKEN, width=50)
        f.pack(side=TOP)
        l.pack(side=LEFT)
        e.pack(side=RIGHT)

if __name__ == '__main__':
    root = Tk()
    gridbox(Toplevel())
    packbox(Toplevel())
    Button(root, text='Quit', command=root.quit).pack()
    mainloop()
```

Эти две функции создают графические элементы меток и полей ввода одинаковым способом, но размещаются они совершенно разными путями:

- При использовании `pack` метки и поля ввода прикрепляются к левому и правому краям с помощью параметров `side`, и для каждого ряда создается `Frame` (котрый прикрепляется к верхнему краю родителя).
- При использовании `grid` каждому графическому элементу назначается положение с помощью параметров `row` (ряд) и `column` (колонка) в предполагаемой табличной сетке родителя.

Разница в объеме кода, необходимого для каждой схемы, ничтожна: схема с `pack` должна создавать `Frame` для каждого ряда, а схема с `grid` должна отслеживать номер текущего ряда. При выполнении сценария создаются окна, показанные на рис. 8.26.

Рис. 8.26. Эквивалентные окна для grid и pack

Сочетание grid с pack

Обратите внимание, что в предыдущем примере в каждую функцию конструктора формы передается новое значение `Toplevel`, благодаря чему версии `grid` и `pack` приводят к различным окнам верхнего уровня. Так как два менеджера геометрии взаимно исключают друг друга в заданном родительском окне, необходимо следить за тем, чтобы по недосмотру не смешать их. Пример 8.19 показывает, что графические элементы можно упаковывать и располагать по сетке в одном и том же окне, но только заключив их в отдельные контейнеры `Frame`.

Пример 8.19. PP2E\Gui\Tour\Grid\grid2-same.py

```
##########################################################################
# нельзя выполнять grid и pack в одном родительском контейнере (например, корневом окне)
# но можно смешивать в одном окне, помещая в разные родительские фреймы;
##########################################################################
from Tkinter import *
from grid2 import gridbox, packbox
root = Tk()
Label(root, text='Grid:').pack()
frm = Frame(root, bd=5, relief=RAISED); frm.pack(padx=5, pady=5)
gridbox(frm)
Label(root, text='Pack:').pack()
frm = Frame(root, bd=5, relief=RAISED); frm.pack(padx=5, pady=5)
packbox(frm)
Button(root, text='Quit', command=root.quit).pack()
mainloop()
```

При выполнении получится составное окно с двумя формами идентичного вида (рис. 8.27), но эти два вложенные фрейма в действительности управляются совершенно разными менеджерами геометрии.

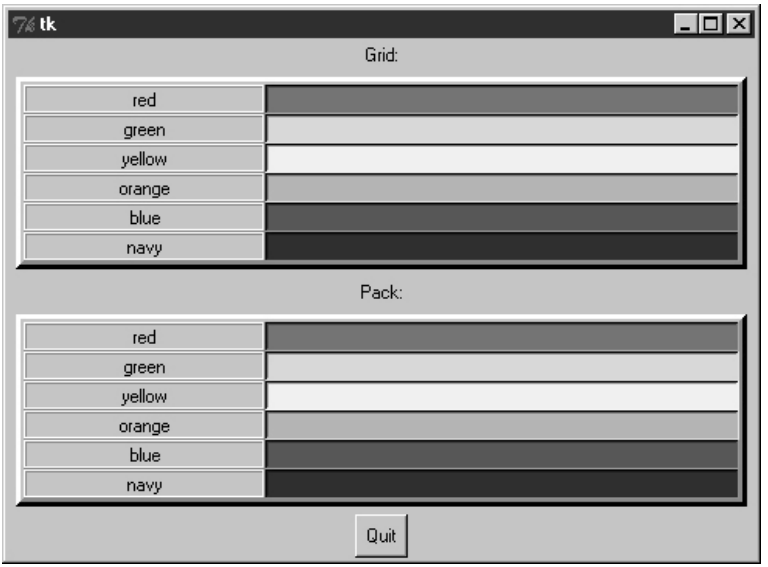


Рис. 8.27. grid и pack в одном окне

Напротив, код типа приведенного в примере 8.20 вызывает грубую ошибку, поскольку пытается выполнять `pack` и `grid` в одном и том же родителе – только один менеджер геометрии может использоваться в каждом отдельном родительском окне.

Пример 8.20. PP2E\Gui\Tour\Grid\grid2-fails.py

```
######################################################################
# ОТКАЗ-- нельзя использовать grid и pack в одном родителе (корневом окне)
######################################################################
from Tkinter import *
from grid2 import gridbox, packbox
root = Tk()
gridbox(root)
packbox(root)
Button(root, text='Quit', command=root.quit).pack()
mainloop()
```

Этот сценарий передает одного и того же родителя (окно верхнего уровня) каждой из функций, пытаясь вывести обе формы в одном окне. На моей машине он полностью подвешивает процесс Python, не выводя вообще никаких окон (в Windows 98 мне пришлось прибегнуть к <Ctrl>+<Alt>+<Delete>, чтобы убить его). Комбинирование менеджеров геометрии может представлять сложность, пока с этим не освоишься; скажем, чтобы заработал этот пример, требуется просто изолировать окно с сеткой в собственном родительском контейнере, чтобы оградить его от упаковки, выполняемой в корневом окне:

```
root = Tk()
frm = Frame(root)
frm.pack()            # это работает
gridbox(frm)          # у gridbox должен быть собственный родитель
packbox(root)
Button(root, text='Quit', command=root.quit).pack()
mainloop()
```

Еще раз напоминаем, что в настоящее время внутри одного родителя обязательно применять либо pack, либо grid, но не тот и другой одновременно. Возможно, в будущем это ограничение будет снято, что, впрочем, маловероятно с учетом различий в схемах двух менеджеров окон; на всякий случай проверьте свою версию Python.

Обеспечение расширяемости графических элементов, размещаемых по сетке

А теперь немного практики. Сетки, показанные до сих пор, имеют фиксированный размер; они не увеличиваются в размере, когда пользователь изменяет размер содержащего их окна. Пример 8.21 снова реализует чрезвычайно патриотическую форму ввода с обоими методами, grid и pack, но в нем есть дополнительные средства, необходимые для того, чтобы все графические элементы в обоих окнах расширялись вместе со своими окнами.

Пример 8.21. PP2E\Gui\Tour\Grid\grid3.py

```
# добавлены метки и изменение размеров

from Tkinter import *
colors = ['red',  'white',  'blue']

def gridbox(root):
    Label(root, text='Grid').grid(columnspan=2)
    r = 1
    for c in colors:
        l = Label(root, text=c, relief=RIDGE,  width=25)
        e = Entry(root, bg=c,   relief=SUNKEN, width=50)
        l.grid(row=r, column=0, sticky=NSEW)
        e.grid(row=r, column=1, sticky=NSEW)
        root.rowconfigure(r, weight=1)
        r = r+1
    root.columnconfigure(0, weight=1)
    root.columnconfigure(1, weight=1)

def packbox(root):
    Label(root, text='Pack').pack()
    for c in colors:
        f = Frame(root)
        l = Label(f, text=c, relief=RIDGE,  width=25)
        e = Entry(f, bg=c,   relief=SUNKEN, width=50)
        f.pack(side=TOP,   expand=YES, fill=BOTH)
        l.pack(side=LEFT,  expand=YES, fill=BOTH)
        e.pack(side=RIGHT, expand=YES, fill=BOTH)
root = Tk()
gridbox(Toplevel(root))
packbox(Toplevel(root))
Button(root, text='Quit', command=root.quit).pack()
mainloop()
```

При запуске этого сценария показывается то, что изображено на рис. 8.28. Снова создаются отдельные окна для упаковки и сетки с полями ввода в правой части, окрашенными в красный, белый и голубой цвета (или для читателей, которые не работают параллельно на компьютере: серый, белый и спорно более темный серый).

Однако на этот раз изменение размеров обоих окон с помощью перетаскивания мышью заставляет все встроенные в них метки и поля ввода расширяться вместе с окнами, как показано на рис. 8.29.

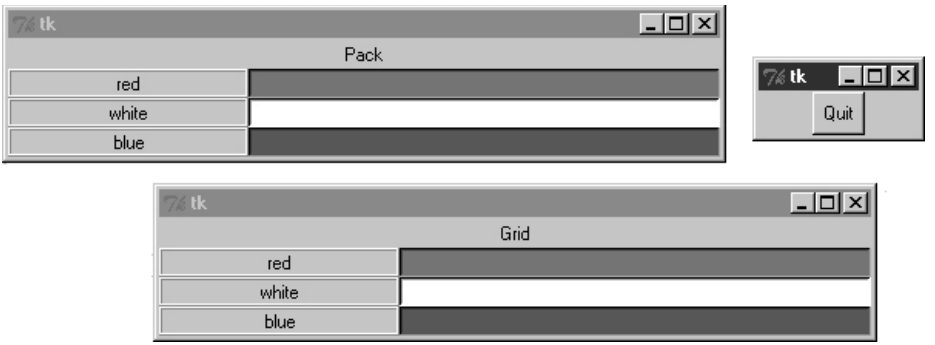


Рис. 8.28. Окна grid и pack до изменения размеров

Рис. 8.29. Окна grid и pack после изменения размера

Изменение размеров в сетках

Теперь, когда я показал, что делают эти окна, нужно объяснить, как они это делают. Ранее мы узнали, как заставить графические элементы расширяться при использовании `pack`: с помощью параметров `expand` и `fill` увеличивается отводимое им пространство, в котором они растягиваются. Чтобы расширение действовало для элементов, размещаемых посредством `grid`, требуется использовать другие протоколы: ряды и колонки становятся расширяемыми, когда они помечены с помощью *weight* (вес), а элементы растягиваются в отведенных им ячейках сетки, когда помечены с помощью *sticky* (липкий):

Тяжелые ряды и колонки

При использовании `pack` ряды становятся расширяемыми, если расширяемым делается соответствующий `Frame` в результате задания `expand=YES` и `fill=BOTH`. Для сетки нужно быть несколько конкретнее: чтобы добиться полной расширяемости, требуется вызвать для контейнера сетки метод `rowconfig` для каждого ряда и метод

columnconfig для каждой колонки. Обоим методам нужно передать параметр веса со значением, большим нуля, чтобы ряды и колонки стали расширяемыми. По умолчанию вес принимается равным нулю (что означает отсутствие расширения), а контейнером сетки в данном сценарии служит просто окно верхнего уровня. Использование разных весов для разных рядов и колонок заставляет их расширяться в различных пропорциях.

Липкие элементы

При использовании pack графические элементы растягиваются по горизонтали или вертикали, заполняя отведенное им пространство, если задать параметр fill, а для позиционирования элементов в отведенном им пространстве используется параметр anchor. Для grid параметр sticky выполняет роли обоих параметров – fill и anchor – из упаковщика. Графические элементы, размещаемые по сетке, можно прилепить к одному краю отведенной им клетки (как с помощью anchor) или более, чем к одному краю, чтобы заставить их растягиваться (как с помощью fill). Графические элементы можно приклеивать в четырех направлениях – N, S, E и W, а комбинируя эти четыре буквы можно задавать приклеивание к нескольким сторонам. Например, установка sticky в W выравнивает графический элемент по левому краю отведенного ему пространства (подобно anchor=W в упаковщике), а NS растягивает графический элемент по вертикали в выделенном пространстве (подобно fill=Y в упаковщике).

Приклеивание в графических элементах не использовалось в предыдущих примерах, потому что в структурах участвовали обычные размеры (элементы были не меньше пространства, отведенного им в ячейке сетки), а изменение размеров вообще не поддерживалось. В данном случае в сценарии задана липкость NSEW, чтобы графические элементы растягивались во всех направлениях вместе с отведенными им ячейками.

Различные сочетания весов рядов и колонок, а также значения липкости создают различные эффекты при изменении размеров. Например, удаление строк с columnconfig из сценария grid3 заставляет изображение расширяться в вертикальном направлении, но не в горизонтальном. Попробуйте сами изменить некоторые из этих установок и посмотреть, к каким эффектам это приведет.

Слияние колонок или рядов

Есть еще одно большое отличие в том, как сценарий grid3 конфигурирует свои окна. Оба окна – с grid и pack – выводят вверху метку на ширину всего окна. В схеме с упаковкой просто создается метка, прикрепленная к верхнему краю окна в целом (вспомните, что side по умолчанию имеет значение TOP):

```
Label(root, text='Pack').pack()
```

Так как эта метка прикрепляется к верхней части окна раньше, чем фреймы рядов, она, как и требовалось, охватывает весь верх окна. Однако размещение такой метки в строгом мире сеток требует несколько большей работы; в первой строке функции, реализующей сетку, это делается следующим образом:

```
Label(root, text='Grid').grid(columnspan=2)
```

Для того чтобы графический элемент охватывал несколько колонок, методу grid передается параметр columnspan, указывающий количество охватываемых колонок. В данном случае он просто указывает, что метка вверху окна должна простираться на все окно, охватывая и колонку с метками, и колонку с полями ввода. Если нужно, чтобы графический элемент охватывал несколько рядов, следует передать параметр

rowspan. Правильная структура сеток может быть преимуществом или недостатком в зависимости от того, насколько правильным должен быть интерфейс пользователя; эти два параметра для установки диапазонов позволяют при необходимости делать исключения из правила.

Так какой же менеджер геометрии оказывается здесь победителем? Если имеет значение изменение размеров, как в этом сценарии, то использование сетки в действительности оказывается несколько *более* сложным (в самом деле в данном примере для сетки потребовалось три дополнительных строки кода). С другой стороны, grid прекрасно служит в простых формах, а ваши сетки и упаковки могут быть другими.

Создание с помощью grid более крупных таблиц

До сих пор мы строили наборы меток и полей ввода из двух колонок. Это типичный вид форм для ввода, но в Tkinter менеджер grid способен организовывать значительно более крупные матрицы. В примере 8.22 создается массив меток, состоящий из пяти строк и четырех колонок, в котором каждая метка просто выводит номер своей строки и колонки (row.col). При выполнении этого сценария выводится окно, показанное на рис. 8.30.

Рис. 8.30. Массив 5×4 координат меток

Пример 8.22. PP2E\Gui\Tour\Grid\grid4.py

```
# простая двумерная таблица

from Tkinter import *

for i in range(5):
    for j in range(4):
        l = Label(text='%d.%d' % (i, j), relief=RIDGE)
        l.grid(row=i, column=j, sticky=NSEW)

mainloop()
```

Если вам показалось, что таким способом можно программировать электронные таблицы, то вы, вероятно, напали на правильный след. Пример 8.23 слегка развивает эту мысль и добавляет кнопку, которая выводит в поток stdout текущие значения полей ввода в таблице (обычно в окно консоли).

Пример 8.23. PP2E\Gui\Tour\Grid\grid5.py

```
# двумерная таблица полей ввода

from Tkinter import *

rows = []
for i in range(5):
    cols = []
    for j in range(4):
        e = Entry(relief=RIDGE)
```

```
            e.grid(row=i, column=j, sticky=NSEW)
            e.insert(END, '%d.%d' % (i, j))
            cols.append(e)
        rows.append(cols)

def onPress():
    for row in rows:
        for col in row:
            print col.get(),
        print

Button(text='Fetch', command=onPress).grid()
mainloop()
```

При выполнении этого сценария создается окно, показанное на рис. 8.31, и все графические элементы полей ввода в сетке сохраняются в двумерном списке списков. При нажатии кнопки Fetch сценарий проходит через сохраненный список списков полей ввода, чтобы получить и отобразить все текущие значения в сетке. Вот вывод после двух нажатий Fetch – одного перед изменениями в полях ввода и другого после:

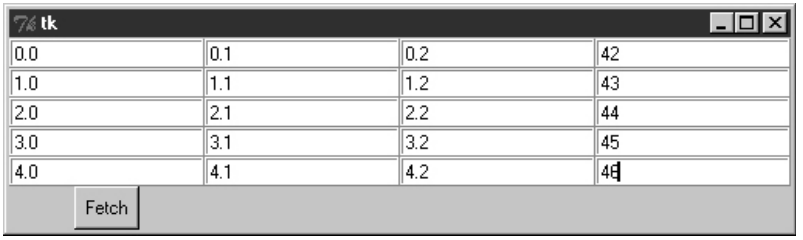


Рис. 8.31. Более крупная сетка полей ввода

```
C:\...\PP2E\Gui\Tour\Grid>python grid5.py
0.0 0.1 0.2 0.3
1.0 1.1 1.2 1.3
2.0 2.1 2.2 2.3
3.0 3.1 3.2 3.3
4.0 4.1 4.2 4.3
0.0 0.1 0.2 42
1.0 1.1 1.2 43
2.0 2.1 2.2 44
3.0 3.1 3.2 45
4.0 4.1 4.2 46
```

Теперь, когда мы знаем, как создавать массивы полей ввода и обходить их, добавим еще несколько полезных кнопок. В примере 8.24 добавлен еще один ряд, в котором выводятся суммы по колонке, и кнопки, которые обнуляют поля и вычисляют суммы по колонкам.

Пример 8.24. PP2E\Gui\Tour\Grid\grid5b.py

```
# добавить суммирование по колонкам и очистку полей

from Tkinter import *
numrow, numcol = 5, 4

rows = []
for i in range(numrow):
    cols = []
```

```
        for j in range(numcol):
            e = Entry(relief=RIDGE)
            e.grid(row=i, column=j, sticky=NSEW)
            e.insert(END, '%d.%d' % (i, j))
            cols.append(e)
        rows.append(cols)

    sums = []
    for i in range(numcol):
        l = Label(text='?', relief=SUNKEN)
        l.grid(row=numrow, col=i, sticky=NSEW)
        sums.append(l)

    def onPrint():
        for row in rows:
            for col in row:
                print col.get(),
            print
        print

    def onSum():
        t = [0] * numcol
        for i in range(numcol):
            for j in range(numrow):
                t[i]= t[i] + eval(rows[j][i].get())
        for i in range(numcol):
            sums[i].config(text=str(t[i]))

    def onClear():
        for row in rows:
            for col in row:
                col.delete('0', END)
                col.insert(END, '0.0')
        for sum in sums:
            sum.config(text='?')

    import sys
    Button(text='Sum',   command=onSum).grid(row=numrow+1, column=0)
    Button(text='Print', command=onPrint).grid(row=numrow+1, column=1)
    Button(text='Clear', command=onClear).grid(row=numrow+1, column=2)
    Button(text='Quit',  command=sys.exit).grid(row=numrow+1, column=3)
    mainloop()
```

На рис. 8.32 показан этот сценарий в действии, суммирующий четыре колонки чисел. Чтобы получить таблицу другого размера, измените переменные `numrow` и `numcol` в начале сценария.

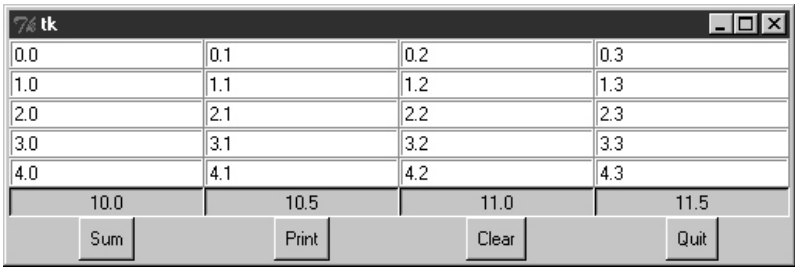


Рис. 8.32. Добавление сумм по колонкам

И наконец, пример 8.25 представляет собой еще одно последнее расширение, код которого оформлен как класс с целью повторного использования, с добавленной в него кнопкой для загрузки таблицы из файла с данными. Предполагается, что файл содержит строку для каждого ряда данных и внутри строки колонки разделяются пробельными символами (пробелами или символами табуляции). При загрузке данных из файла автоматически изменяется размер GUI таблицы, чтобы уместить все колонки.

Пример 8.25. PP2E\Gui\Tour\Grid\grid5c.py

```
# код, преобразованный во встраиваемый класс

from Tkinter import *
from PP2E.Gui.Tour.quitter import Quitter          # повторное использование, pack и grid

class SumGrid(Frame):
    def __init__(self, parent=None, numrow=5, numcol=5):
        Frame.__init__(self, parent)
        self.numrow = numrow                       # я - фрейм, служащий контейнером
        self.numcol = numcol                       # pack или grid применяет вызвавший
        self.makeWidgets(numrow, numcol)           # иначе использовался бы только
                                                   # одним способом

    def makeWidgets(self, numrow, numcol):
        self.rows = []
        for i in range(numrow):
            cols = []
            for j in range(numcol):
                e = Entry(self, relief=RIDGE)
                e.grid(row=i+1, column=j, sticky=NSEW)
                e.insert(END, '%d.%d' % (i, j))
                cols.append(e)
            self.rows.append(cols)

        self.sums = []
        for i in range(numcol):
            l = Label(self, text='?', relief=SUNKEN)
            l.grid(row=numrow+1, col=i, sticky=NSEW)
            self.sums.append(l)

        Button(self, text='Sum',   command=self.onSum).grid(row=0, column=0)
        Button(self, text='Print', command=self.onPrint).grid(row=0, column=1)
        Button(self, text='Clear', command=self.onClear).grid(row=0, column=2)
        Button(self, text='Load',  command=self.onLoad).grid(row=0, column=3)
        Quitter(self).grid(row=0, column=4)    # fails: Quitter(self).pack()

    def onPrint(self):
        for row in self.rows:
            for col in row:
                print col.get(),
            print
        print

    def onSum(self):
        t = [0] * self.numcol
        for i in range(self.numcol):
            for j in range(self.numrow):
                t[i]= t[i] + eval(self.rows[j][i].get())
        for i in range(self.numcol):
            self.sums[i].config(text=str(t[i]))
```

```
    def onClear(self):
        for row in self.rows:
            for col in row:
                col.delete('0', END)
                col.insert(END, '0.0')
        for sum in self.sums:
            sum.config(text='?')

    def onLoad(self):
        import string
        from tkFileDialog import *
        file = askopenfilename()
        if file:
            for r in self.rows:
                for c in r: c.grid_forget()
            for s in self.sums:
                s.grid_forget()
            filelines   = open(file, 'r').readlines()
            self.numrow = len(filelines)
            self.numcol = len(string.split(filelines[0]))
            self.makeWidgets(self.numrow, self.numcol)
            row = 0
            for line in filelines:
                fields = string.split(line)
                for col in range(self.numcol):
                    self.rows[row][col].delete('0', END)
                    self.rows[row][col].insert(END, fields[col])
                row = row+1

if __name__ == '__main__':
    import sys
    root = Tk()
    root.title('Summer Grid')
    if len(sys.argv) != 3:
        SumGrid(root).pack()    # .grid() здесь тоже работает
    else:
        rows, cols = eval(sys.argv[1]), eval(sys.argv[2])
        SumGrid(root, rows, cols).pack()
    mainloop()
```

Обратите внимание, что класс SumGrid из этого модуля избегает применения grid или pack *к себе*. Чтобы дать возможность прикрепления к контейнерам, где есть другие графические элементы, упакованные или размещенные по сетке, он оставляет управление собственной геометрией неопределенным и требует, чтобы вызывающий применял pack или grid к его экземплярам. Контейнеры могут выбрать любую схему для своих дочерних элементов, но прикрепляемым классам компонентов, предназначенным для использования с обоими менеджерами геометрии, нельзя управлять собой, так как они не могут знать заранее политику своего родителя.

Это довольно длинный пример, в котором нет почти ничего нового в отношении расположения по сетке или графических элементов в целом, поэтому я оставлю его, в основном, для самостоятельного чтения и просто покажу, что он делает. На рис. 8.33 показано изначальное окно, созданное этим сценарием, после того как была изменена последняя колонка и произведено суммирование.

По умолчанию класс создает здесь сетку размером 5 на 5, но можно передать другие измерения как в конструкторе класса, так и в командной строке сценария. При нажа-

тии кнопки Load выводится стандартный диалог выбора файла, с которым мы встречались ранее (рис. 8.34).

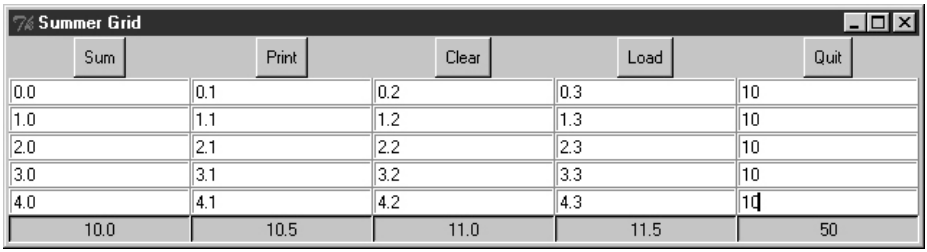


Рис. 8.33. Добавление загрузки данных из файла

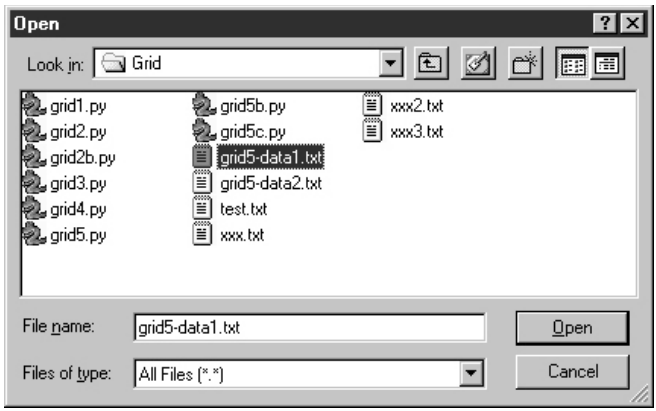


Рис. 8.34. Открытие файла данных для SumGrid

Файл данных *grid-data1.txt* содержит семь строк и шесть колонок данных:

```
C:\...\PP2E\Gui\Tour\Grid>type grid5-data1.txt
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
```

При загрузке его в наш GUI соответствующим образом изменяются размеры сетки – класс просто заново выполняет свою логику создания графических элементов после удаления прежних элементов ввода с помощью метода `grid_forget`.[1] На рис. 8.35 показано, как выглядит экран после загрузки файла.

У файла данных *grid5-data2.txt* те же измерения, но в двух колонках он содержит не просто числа, а выражения. Так как этот сценарий преобразует значения полей ввода с помощью встроенной функции Python `eval`, в полях этой таблицы действителен лю-

[1] `grid_forget` отменяет отображение элементов в сетку и в результате удаляет их с экрана. Посмотрите также методы элемента `pack_forget` и окна `withdraw`, которые применены в событии `after` примеров «будильников» следующего раздела, другие способы удаления и перерисовки компонентов GUI.

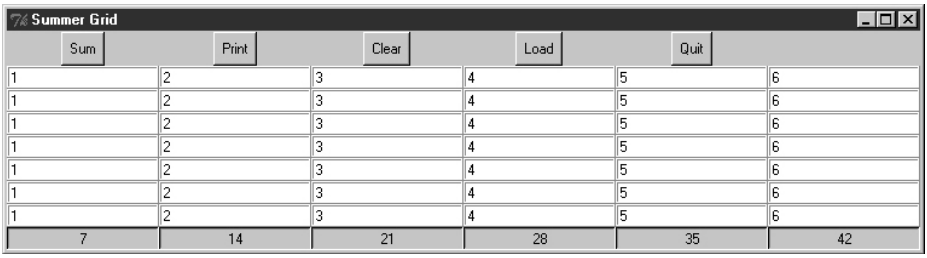


Рис. 8.35. Файл данных загружен, отображен и просуммирован

бой синтаксис Python, если его можно подвергнуть синтаксическому анализу и выполнить в области видимости метода `onSum`:

```
C:\...\PP2E\Gui\Tour\Grid>type grid5-data2.txt
1 2 3 2*2 5 6
1 3-1 3 2<<1 5 6
1 5%3 3 pow(2,2) 5 6
1 2 3 2**2 5 6
1 2 3 [4,3][0] 5 6
1 {'a':2}['a'] 3 len('abcd') 5 6
1 abs(-2) 3 eval('2+2') 5 6
```

При суммировании этих полей выполняется содержащийся в них код Python, что иллюстрирует рис. 8.36. Это может оказаться мощной функцией. Представьте себе, например, полноценную сетку электронной таблицы – значения полей могут быть «фрагментами» кода Python, которые динамически вычисляют значения, вызывают функции из модулей и даже загружают текущие котировки акций через Интернет с помощью инструментов, с которыми мы познакомимся в следующей части книги.

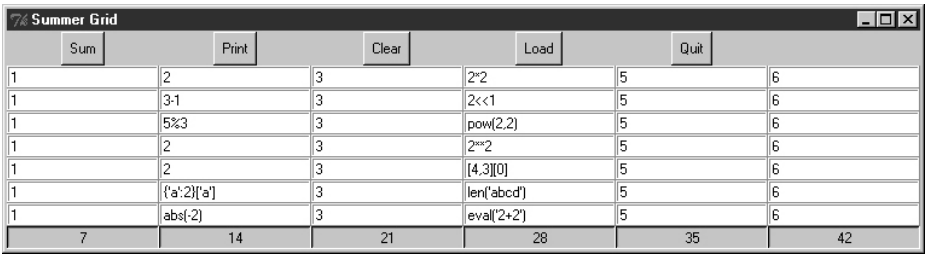


Рис. 8.36. Выражения Python в данных и таблице

Эта программа может оказаться потенциально опасной – в поле может содержаться выражение, удаляющее содержимое вашего жесткого диска! Если вы не очень уверены в том, что могут сделать выражения, не используйте `eval` (осуществляйте преобразование с помощью функций с большими ограничениями типа `int` и `float`) или почитайте в главе 15 о модуле Python `rexec` для установления режима ограниченного выполнения.

Конечно, это еще далеко от подлинной программы электронных таблиц. Дальнейшие изменения, ведущие к этой цели, оставляются в качестве упражнения для читателя. Я хочу также отметить, что о размещении по сетке можно сказать больше, чем позволяет имеющееся место. Например, путем создания вложенных фреймов с собственными сетками можно строить более сложные структуры во многом аналогично тому, как размещает вложенные фреймы упаковщик. А теперь перейдем к последней теме обзора графических элементов.

Средства синхронизации, потоки и анимация

Последняя остановка в нашей экскурсии по графическим элементам самая необычная. Вместе с Tkinter поставляется ряд средств для поддержки модели программирования, управляемого событиями, а не отображения графики на экране.

Некоторым приложениям GUI требуется периодически выполнять действия в фоновом режиме. Например, для того чтобы графический элемент имел «мерцающий» вид, хотелось бы зарегистрировать обработчик обратного вызова, который вызывался бы через равные промежутки времени. Аналогично позволить длительной операции с файлом заблокировать прочие действия в GUI было бы не очень хорошим решением. Если бы удалось заставить цикл событий периодически обновляться, GUI мог бы остаться активным. В Tkinter есть средства для планировки таких отложенных действий и принудительного обновления экрана:

`widget.after(`*`milliseconds, function, *args`*`)`

Это средство планирует вызов заданной функции по истечении заданного числа миллисекунд. `function` может представлять собой любой вызываемый объект Python: функцию, связанный метод и т. д. В такой форме этот вызов не останавливает программу – функция обратного вызова запускается позднее из обычного цикла событий Tkinter. Количество миллисекунд может быть задано в виде числа с плавающей точкой и указывать доли секунды. Вызов возвращает ID, который можно передать `after_cancel`, чтобы отменить обратный вызов. Этот метод используется очень часто, поэтому несколько ниже о нем будет сказано подробнее и приведен пример.

`widget.after(`*`milliseconds`*`)`

Этот инструмент останавливает выполнение программы на заданное количество миллисекунд. Например, аргумент, равный 5000, останавливает программу на 5 секунд. В сущности, это то же самое, что библиотечная функция Python `time.sleep`; оба вызова могут применяться для создания задержки при отображении экранов с учетом времени (например, в анимационных программах типа PyDraw и более простых примерах ниже).

`widget.after_idle(`*`function, *args`*`)`

Это средство планирует вызов заданной функции при отсутствии других событий, которые должны обрабатываться. Таким образом, `function` становится обработчиком холостого времени, вызываемым, когда GUI не занят ничем другим.

`widget.after_cancel(`*`id`*`)`

Этот инструмент отменяет текущее событие обратного вызова `after` до того, как оно произойдет.

`widget.update()`

Этот инструмент вынуждает Tkinter обработать все текущие события, имеющиеся в очереди, в том числе изменение геометрии и обновление и перерисовку графических элементов. Его можно периодически вызывать из долго выполняющегося обработчика обратного вызова, чтобы обновить экран и произвести те изменения, которые уже запросил ваш обработчик. Если этого не делать, произведенные обработчиком изменения появятся на экране только после выхода из него. В действительности на время работы долгого обработчика экран может вообще зависнуть, если не обновлять его вручную (обработчики не выполняются в отдельных потоках, о чем говорится в следующем разделе); окно даже не будет перерисовывать себя при накрытии или открытии другими окнами, пока не произойдет возврат из обработчика. Например, программы, осуществляющие анимацию путем повторения перемещения объекта и остановки, должны делать этот вызов, не дожидаясь

конца анимации, иначе на экране можно будет увидеть только конечное положение объекта. Что еще хуже, GUI окажется совершенно бездейственным, пока не произойдет возврат из обратного вызова анимации (смотрите простые примеры анимации далее в этой главе и PyDraw в следующей главе).

`widget.update_idletasks()`

Этот инструмент обрабатывает все имеющиеся события для холостого времени. Иногда он безопаснее, чем `after`, который в некоторых случаях может стать причиной возникновения состояния гонки[1] (race conditions).

`_tkinter.createfilehandler(`*`file, mask, function`*`)`

Этот инструмент назначает функцию, которая будет вызываться при изменении статуса файла. Функция может быть вызвана, если в файле появляются данные для чтения, он становится доступным для записи или возбуждает исключительную ситуацию. Обработчики файлов часто используются для обработки каналов и сокетов, так как обычные запросы ввода/вывода могут блокировать вызвавшего. Для Windows этот вызов в Tk 8.0 не действует и потому не будет рассматриваться в данной книге.

```
widget.wait_variable(var)
widget.wait_window(win)
widget.wait_visibility(win)
```

Эти инструменты приостанавливают вызвавшего до момента, когда переменная Tkinter изменит свое значение, будет разрушено окно или окно станет видимым. Все они входят в локальный цикл событий, поэтому `mainloop` приложения продолжает обработку событий. Обратите внимание, что `var` является объектом переменной Tkinter (обсуждавшимся ранее), а не простой переменной Python. Для использования в модальных диалогах сначала вызовите `widget.focus()` (чтобы установить фокус ввода) и `widget.grab()` (чтобы сделать окно единственным активным).

Мы не станем здесь более подробно вникать во все эти средства. За дополнительной информацией обратитесь к другой документации по Tk и Tkinter.

Использование потоков с GUI

Следует иметь в виду, что во многих программах реализованная в Python поддержка потоков, с которой мы познакомились в главе 3, может отчасти выполнять те же роли, что и инструменты Tkinter, перечисленные в предыдущем разделе. Например, чтобы избежать блокировки GUI во время долгих операций обмена данными через файлы или сокеты, этот обмен можно осуществлять в порожденных потоках при обычном выполнении остальной программы. С такими многопоточными программами GUI мы встретимся в части III «Сценарии для Интернета» (например, PyMailGui в главе 11 «Сценарии, выполняемые на стороне клиента»). Аналогично GUI, ожидающие ввода из каналов или сокетов, могут делать это в порожденных потоках (или обратных вызовах `after`), не блокируя собственно GUI.

Однако при использовании потоков в программах Tkinter вызовы GUI может осуществлять только *главный* поток (тот, в котором построен GUI и запущен `mainloop`). Даже такие вещи, как вызов метода `update`, описанного в предыдущем разделе, нельзя выполнять из порожденных потоков программы GUI – они в самый неожиданный мо-

[1] Термин, означающий ситуацию, когда два различных процесса или потока могут одновременно захватить контроль над одним и тем же ресурсом, требующим поочередного доступа к себе. В результате ресурс может оказаться поврежденным (например, одновременная модификация двумя процессами одной записи в базе данных). – *Примеч. науч. ред.*

мент могут приводить к краху программы. В будущих версиях Python и Tkinter такое отношение GUI к потокам может улучшиться, но на сегодняшний день оно налагает некоторые структурные и специфические для платформ ограничения.

Например, так как порожденные потоки не могут выполнять обработку GUI, они обычно должны поддерживать связь с главным потоком посредством глобальных переменных, если это требуется приложению. Скажем, поток, ждущий данных из сокета, может просто устанавливать глобальные переменные, которые инициируют изменения в GUI через обработчики обратного вызова события `after`. Обратите внимание, что это ограничение обусловлено не Python или Tkinter (оно располагается значительно ниже в иерархии программного обеспечения, выполняющего ваш GUI) и может исчезнуть в будущем. Кроме того, некоторые вызовы для холста в Tkinter фактически могут быть безопасными для потоковых переменных (thread-safe) (см. сценарий анимации в примере 8.31). Мы еще вернемся далее к этому ограничению, когда встретимся с более крупными многопоточными программами GUI.

Применение метода after

Метод `after` позволяет сценариям назначать обработчик обратного вызова, который будет выполнен в некоторый момент времени в будущем, и мы часто будем пользоваться им в последующих примерах. В частности, в главе 9 мы познакомимся с программой *clock*, которая с помощью `after` просыпается 10 раз в секунду и получает текущее время, и с программой показа слайдов *slideshow*, которая с помощью `after` устанавливает время показа новой фотографии (см. PyClock и PyView). Для иллюстрации основ планирования обратных вызовов служит пример 8.26.

Пример 8.26. PP2E\Gui\Tour\alarm.py

```
#!/usr/local/bin/python
from Tkinter import *

class Alarm(Frame):
    def repeater(self):                          # каждые N миллисекунд
        self.bell()                              # сигнал
        self.stopper.flash()                     # мигающая кнопка
        self.after(self.msecs, self.repeater)   # переустановить обработчик
    def __init__(self, msecs=1000):              # по умолчанию = 1 секунда
        Frame.__init__(self)
        self.msecs = msecs
        self.pack()
        stopper = Button(self, text='Stop the beeps!', command=self.quit)
        stopper.pack()
        stopper.config(bg='navy', fg='white', bd=8)
        self.stopper = stopper
        self.repeater()

if __name__ == '__main__': Alarm(msecs=1000).mainloop()
```

Этот сценарий создает окно, показанное на рис. 8.37, и периодически вызывает метод кнопки `flash`, заставляющий кнопку на мгновение вспыхнуть (быстро чередует ее цвета), и метод Tkinter `bell`, обращающийся к интерфейсу системы для звука. Метод `repeater` вызывает сигнал со вспышкой и устанавливает обратный вызов, который будет выполнен через заданный промежуток времени, с помощью метода `after`.

Однако `after` не влечет остановки вызвавшего: обратные вызовы происходят в фоновом режиме, в то время как программа производит другую обработку – технически с того самого момента, когда цикл событий Tk получит возможность обнаружить изме-

нение времени. Для осуществления этого `repeater` каждый раз вызывает `after` и заново назначает обратный вызов. Отложенные события являются *одноразовыми*: чтобы повторить событие, нужно снова его запланировать.

Рис. 8.37. Прекратите пищать!

В итоге при выполнении этот сценарий начинает подавать звуковые сигналы и вспышки сразу, как только будет выведено его окно с одной кнопкой, и продолжает сигналить и вспыхивать, сигналить и вспыхивать. Прочие действия и операции GUI не влияют на это. Даже если свернуть окно, сигналы продолжаются, потому что события таймера Tkinter генерируются в фоновом режиме. Чтобы прекратить сигналы, нужно убить окно или нажать на кнопку. Изменив задержку `msecs`, можно заставить сигнал звучать чаще или реже, насколько позволит система (максимально допустимая частота может зависеть от платформы). Предупреждаю заранее, что это не лучшая демонстрационная программа для запуска в многолюдном помещении.

Скрытие и перерисовка графических элементов и окон

Метод кнопки `flash` вызывает вспышку кнопки, но нетрудно динамически изменять другие параметры внешнего вида таких графических элементов, как кнопки, метки и текст, с помощью метода графических элементов `config`. Например, можно добиться такого же эффекта типа вспышки путем инвертирования вручную цветов переднего и заднего плана элементов с помощью метода `config` в обработчиках обратного вызова, установленных методом `after`. Исключительно для развлечения пример 8.27 содержит код для подачи сигнала, в котором сделан еще один шаг.

Пример 8.27. PP2E\Gui\Tour\alarm-hide.py

```
from Tkinter import *
import alarm

class Alarm(alarm.Alarm):                          # изменить обратный вызов таймера
    def repeater(self):                            # каждые N миллисекунд
        self.bell()                                # подать сигнал
        if self.shown:
            self.stopper.pack_forget()             # спрятать кнопку
        else:                                      # или изменить цвета...
            self.stopper.pack()
        self.shown = not self.shown                # переключить состояние для следующего раза
        self.after(self.msecs, self.repeater)     # переустановить обработчик
    def __init__(self, msecs=1000):                # по умолчанию = 1 секунда
        self.shown = 0
        alarm.Alarm.__init__(self, msecs)

if __name__ == '__main__': Alarm(msecs=500).mainloop()
```

При выполнении этого сценария появляется то же самое окно, но теперь кнопка поочередно стирается или вновь отображается в каждом событии таймера. Метод графического элемента `pack_forget` стирает нарисованный элемент, а `pack` показывает его снова; `grid_forget` и `grid` аналогичным образом скрывают и показывают элементы в сетке. Метод `pack_forget` удобен для динамического изменения работающего GUI. Например, можно решить, какие компоненты должны отображаться, строить графичес-

кие элементы заранее и показывать их только по мере надобности. В данном случае это просто значит, что пользователь должен нажать на кнопку, пока она отображена, иначе шум будет продолжаться.

Чтобы скрывать и показывать *целое окно*, а не просто один элемент внутри него, применяются методы окна верхнего уровня `withdraw` и `deiconify`. Метод `withdraw`, демонстрируемый в примере 8.28, полностью стирает окно и его значок (используйте вместо него `iconify`, если хотите, чтобы при сокрытии окна появлялся его значок), а метод `state` возвращает текущее состояние окна («normal», «iconic» или «withdrawn»). Эти методы можно использовать для динамического показа готовых диалоговых окон, но это, возможно, менее практично.

Пример 8.28. PP2E\Gui\Tour\alarm-withdraw.py

```
from Tkinter import *
import alarm

class Alarm(alarm.Alarm):
    def repeater(self):                             # каждые N миллисекунд
        self.bell()                                 # сигнал
        if self.master.state() == 'normal':         # окно видно?
            self.master.withdraw()                  # скрыть все окно, без значка
        else:                                       # iconify свертывает к значку
            self.master.deiconify()                 # иначе заново нарисовать все окно
            self.master.lift()                      # и поднять его на передний план
        self.after(self.msecs, self.repeater)      # переназначить обработчик

if __name__ == '__main__': Alarm().mainloop()      # владелец = корень Tk по умолчанию
```

Этот сценарий действует так же за исключением того, что при сигнале появляется или исчезает все окно – нажимать надо тогда, когда его видно. Можно добавить к этому таймеру массу других эффектов. Будут ли ваши кнопки мерцать и исчезать зависит, скорее, от терпения пользователей, чем от технологии Tkinter.

Простые технологии анимации

Все GUI, представленные пока в этой книге, за исключением примера `canvasDraw` с непосредственным перемещением фигур, были довольно статичными. В данном, последнем разделе показывается, как можно изменить положение, добавив несколько простых *анимаций* перемещения фигуры в пример 8.15 рисования на холсте. Здесь также демонстрируется понятие *тегов холста* – операции перемещения в этом примере передвигают сразу все объекты на холсте, связанные с тегом. Все овальные фигуры перемещаются при нажатии <O>, а все прямоугольные – при нажатии <R>: как отмечалось ранее, методы, действующие на холсте, принимают как ID объектов, так и имена тегов.

Но главная задача здесь состоит в том, чтобы проиллюстрировать простую технику анимации с помощью инструментов, основанных на времени и описанных выше в этом разделе. Есть три главных способа перемещения объектов по холсту:

- С помощью циклов, использующих `time.sleep` для остановки на доли секунды между последовательными операциями перемещения наряду с вызовами `update` вручную. Сценарий выполняет перемещение, засыпает, передвигает объект еще немного и т. д. Вызов `time.sleep` приостанавливает вызвавшего и потому не возвращает управление в цикл событий GUI – выполнение новых запросов, поступающих во время перемещения, откладывается. Из-за этого после каждого перемещения нужно вызывать `canvas.update`, чтобы перерисовать экран, иначе обновления

экрана не произойдет, пока не закончится весь цикл перемещения в обратном вызове и не осуществится возврат. Это классическая схема долго выполняющегося обратного вызова. Без вызова обновления экрана вручную никакие другие события GUI не будут обработаны до возврата из обратного вызова (даже перерисовка окна).

- С помощью метода `widget.after`, планирующего выполнение операций перемещения через каждые несколько миллисекунд. Поскольку этот подход основан на расписании событий, которые Tkinter отправляет обработчикам, он допускает параллельное осуществление нескольких перемещений и не требует вызовов `canvas.update`. Для выполнения перемещений используется цикл событий, поэтому паузы для сна не нужны и GUI не блокируется во время осуществления перемещений.
- С помощью потоков, в которых выполняется несколько экземпляров циклов с остановкой `time.sleep` из первого подхода. Так как потоки выполняются параллельно, сон в любом из потоков не блокирует ни GUI, ни другие перемещающие потоки. GUI в целом не нужно обновлять из порожденных потоков (на самом деле вызов `canvas.update` из порожденного потока в текущих версиях может привести к краху GUI), но некоторые вызовы холста, в частности перемещение, могут быть безопасными для потоковых переменных в текущей реализации.

Из этих трех схем самые плавные анимации дает первая, но она несколько замедляет другие операции во время перемещения; вторая схема дает несколько более замедленное перемещение, чем остальные, но в целом безопаснее, чем использование потоков; обе последние схемы позволяют одновременно передвигать несколько объектов.

Использование циклов time.sleep

В следующих трех разделах поочередно демонстрируется структура кода для всех трех подходов, создающая новые подклассы примера `canvasDraw`, с которым мы познакомились в примере 8.15. Пример 8.29 иллюстрирует первый подход.

Пример 8.29. PP2E\Gui\Tour\canvasDraw_tags.py

```
##############################################################
# перемещение по тегам посредством time.sleep (не .after или потоков);
# time.sleep не блокирует цикл событий gui во время паузы, но
# не обновляется до выхода из обратного вызова или обращения к .update;
# текущему выполняемому обратному вызову onMove уделяется исключительное внимание,
# пока он не завершит работу: если во время перемещения нажать 'r' или 'o', остальные ждут;
##############################################################

from Tkinter import *
import canvasDraw, time

class CanvasEventsDemo(canvasDraw.CanvasEventsDemo):
    def __init__(self, parent=None):
        canvasDraw.CanvasEventsDemo.__init__(self, parent)
        self.canvas.create_text(75, 8, text='Press o and r to move shapes')
        self.canvas.master.bind('<KeyPress-o>', self.onMoveOvals)
        self.canvas.master.bind('<KeyPress-r>', self.onMoveRectangles)
        self.kinds = self.create_oval_tagged, self.create_rectangle_tagged
    def create_oval_tagged(self, x1, y1, x2, y2):
        objectId = self.canvas.create_oval(x1, y1, x2, y2)
        self.canvas.itemconfig(objectId, tag='ovals', fill='blue')
        return objectId
    def create_rectangle_tagged(self, x1, y1, x2, y2):
        objectId = self.canvas.create_rectangle(x1, y1, x2, y2)
        self.canvas.itemconfig(objectId, tag='rectangles', fill='red')
```

```
        return objectId
    def onMoveOvals(self, event):
        print 'moving ovals'
        self.moveInSquares(tag='ovals')          # переместить все включенные в тег овалы
    def onMoveRectangles(self, event):
        print 'moving rectangles'
        self.moveInSquares(tag='rectangles')
    def moveInSquares(self, tag):                # 5 повторений по 4 раза в секунду
        for i in range(5):
            for (diffx, diffy) in [(+20, 0), (0, +20), (-20, 0), (0, -20)]:
                self.canvas.move(tag, diffx, diffy)
                self.canvas.update()             # принудительное обновление экрана
                time.sleep(0.25)                 # пауза, не блокирующая gui

if __name__ == '__main__':
    CanvasEventsDemo()
    mainloop()
```

Все три сценария этого раздела при вытягивании новых фигур с помощью левой кнопки мыши создают окно с голубыми овалами и красными прямоугольниками. Реализация вытягивания наследуется у суперкласса. Щелчок правой кнопкой мыши немедленно перемещает одну фигуру, а двойной щелчок левой кнопкой также очищает холст – эти операции унаследованы от суперкласса. В действительности в этом новом сценарии лишь изменены вызовы, создающие объекты, в которые добавлены теги и цвета, добавлено текстовое поле и привязки и обратные вызовы для перемещения. На рис. 8.38 показано, как выглядит окно этого подкласса после создания нескольких фигур для анимации.

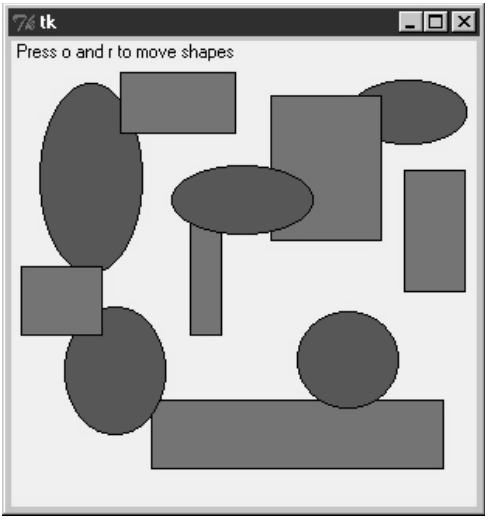


Рис. 8.38. Нарисованные объекты готовы к анимации

С помощью клавиш <O> и <R> начинается анимация всех нарисованных овалов и прямоугольников соответственно. Например, при нажатии <O> начинают перемещаться все голубые овалы. При анимации объекты вычерчивают пять квадратов вокруг своего местоположения, перемещаясь четыре раза в секунду. Новые объекты, которые вычерчиваются, когда другие находятся в движении, тоже начинают перемещаться, потому что помечены тегами. Конечно, нужно запустить этот сценарий,

чтобы получить представление о простой анимации, которую он реализует (правда, можно попытаться подвигать влево-вправо и вверх-вниз книгу, но это не совсем то, что нужно, и в общественном месте может выглядеть глупо).

Использование событий .after

Главный недостаток первого подхода в том, что одновременно может происходить только одна анимация: если нажать <R> или <O> во время движения, новый запрос приостанавливает предыдущее перемещение до своего окончания, потому что каждый обработчик обратного вызова для перемещения допускает только один поток управления при своей работе. Обновление экрана тоже происходит несколько замедленно во время перемещений, поскольку случается только тогда, когда вручную выполняются вызовы `update` (попробуйте вытащить фигуру или закрыть/открыть окно во время перемещения и вы в этом убедитесь сами). Пример 8.30 уточняет метод `moveInSquares`, чтобы снять такие ограничения.

Пример 8.30. PP2E\Gui\Tour\canvasDraw_tags_after.py

```
##############################################################################
# аналогично, но с планируемыми событиями.after, а не циклами time.sleep;
# так как это планируемые события, _одновременно_ могут перемещаться и
# овалы и прямоугольники, не требуя вызовов update для обновления gui
# (только один обратный вызов с циклом time.sleep может выполняться
# одновременно, блокируя при этом другие до своего завершения);
# движение станет беспорядочным, если еще раз нажать 'o' или 'r'
# --несколько обновлений перемещения начнут возникать примерно в одно время;
##############################################################################

from Tkinter import *
import canvasDraw_tags

class CanvasEventsDemo(canvasDraw_tags.CanvasEventsDemo):
    def moveEm(self, tag, moremoves):
        (diffx, diffy), moremoves = moremoves[0], moremoves[1:]
        self.canvas.move(tag, diffx, diffy)
        if moremoves:
            self.canvas.after(250, self.moveEm, tag, moremoves)
    def moveInSquares(self, tag):
        allmoves = [(+20, 0), (0, +20), (-20, 0), (0, -20)] * 5
        self.moveEm(tag, allmoves)

if __name__ == '__main__':
    CanvasEventsDemo()
    mainloop()
```

В этой версии можно одновременно перемещать овалы и прямоугольники – чтобы увидеть это, нарисуйте несколько овалов и прямоугольников, а затем нажмите клавишу <O> и сразу затем клавишу <R>. Попробуйте нажать обе клавиши несколько раз; чем больше нажатий, тем интенсивнее движение, потому что генерируется много событий, перемещающих объекты из того места, в котором они находятся. Если во время перемещения нарисовать новую фигуру, она, как и раньше, немедленно начнет перемещаться.

Использование нескольких потоков циклов time.sleep

Выполнением анимации в потоках можно иногда добиться того же эффекта. В целом обновлять экран из порожденного потока опасно, но в данном примере это действует, по крайней мере в Windows. В примере 8.31 каждая задача анимации выполняется

как независимый и параллельный поток. Это означает, что при каждом нажатии клавиш <O> или <R> для начала анимации порождается новый поток, который выполняет эту задачу. Такая схема работает в Windows, но под Linux она потерпела у меня неудачу – экран не обновляется при изменении его в потоках, и никаких изменений не будет видно до возникновения в дальнейшем событий GUI.

Пример 8.31. PP2E\Gui\Tour\canvasDraw_tags_thread.py

```
##############################################################################
# аналогично, но циклы time.sleep выполняются параллельно в разных потоках,
# а не в событиях .after или одном действующем цикле time.sleep; так как потоки
# выполняются параллельно, здесь также можно перемещать овалы и прямоугольники
# одновременно, и не требуются вызовы для обновления gui: на самом деле вызов
# .update() в настоящее время может привести к краху, хотя некоторые вызовы холста,
# по-видимому, безопасны для использования в потоках, иначе все это вообще не работало бы.
##############################################################################

from Tkinter import *
import canvasDraw_tags
import thread, time

class CanvasEventsDemo(canvasDraw_tags.CanvasEventsDemo):
    def moveEm(self, tag):
        for i in range(5):
            for (diffx, diffy) in [(+20, 0), (0, +20), (-20, 0), (0, -20)]:
                self.canvas.move(tag, diffx, diffy)
                time.sleep(0.25)                    # пауза только в этом потоке
    def moveInSquares(self, tag):
        thread.start_new_thread(self.moveEm, (tag,))

if __name__ == '__main__':
    CanvasEventsDemo()
    mainloop()
```

В этой версии возможно одновременное перемещение фигур, как и в примере 8.30, но на этот раз оно отражает параллельное выполнение потоков. На самом деле используется та же схема, что и в первой версии `time.sleep`. Однако в данном случае активных потоков управления может быть несколько, поэтому обработчики перемещений могут перекрываться во времени – `time.sleep` блокирует только вызвавший поток, а не программу в целом. Такая схема оказывается работоспособной (по крайней мере в Windows), но обычно надежнее использовать потоки только для вычислений, а какое-либо обновление экрана производить в главном потоке (создавшем GUI). Не исключено, что потоки GUI будут лучше поддерживаться в следующих версиях Tkinter, поэтому ищите дополнительные сведения в новых выпусках.

Другие варианты анимации

Мы снова обратимся к анимации в примере PyDraw в главе 9. В нем будут возрождены все три приема, чтобы перемещать фигуры, текст и фотографии в произвольные точки холста, помечаемые щелчком мыши. И хотя система абсолютных координат графического элемента canvas делает его рабочей лошадкой большинства нетривиальных анимаций, в целом возможности анимации в Tkinter ограничены, в основном, вашим воображением. Как было показано ранее в примерах `alarm`, где использовались скрытие графических элементов и вспышки, можно легко динамически изменять внешний вид и других графических элементов. Можно даже динамически удалять и перерисовывать графические элементы и окна.

Должен также заметить, что некоторые приемы перемещения и анимации, показанные в этой и следующей главах, пригодны для многих игровых программ, хотя и не для всех. Если требуется более сложная трехмерная анимация, следует обратить внимание на поддержку пакетом расширения PIL распространенных форматов файлов анимации и фильмов, таких как FLI и MPEG. Python, в его текущей реализации, не часто используется в качестве единственного языка для реализации игровых программ, интенсивно использующих графику, но его все же можно применять в качестве языка, на котором пишутся прототипы и сценарии для таких продуктов.[1] А при интеграции с библиотеками 3D-графики его роль может быть еще более расширена. Ссылки на другие имеющиеся расширения для этой области можно найти на *http://www.python.org*.

Конец экскурсии

Здесь будет представлена оболочка для примеров, представленных в обзоре библиотеки Tkinter. Мы увидели все графические элементы и инструменты, предварительный обзор которых был сделан в конце главы 6 «Графические интерфейсы пользователя» (перелистайте книгу назад и найдите краткое описание территории, пройденной в этом путешествии). Дополнительные сведения вы получите, когда все представленные здесь инструменты вновь появятся в более крупных примерах GUI в главе 9 и в оставшейся части книги в целом. Параметры, явным образом здесь не приведенные, можно найти в ресурсах Tk и Tkinter. Они аналогичны тем, которые были представлены в этом обзоре, но место, которое можно отвести для их иллюстрации в данной книге, ограничено как издателем, так и моими возможностями.

Запускающие программы PyDemos и PyGadgets

Завершая главу, я хочу показать реализацию двух GUI, с помощью которых запускаются основные примеры из книги. Последующие два GUI, PyDemos и PyGadgets, служат просто GUI для запуска других GUI-программ. На самом деле мы подошли к концу истории о программах, запускающих демонстрационные примеры: обе программы взаимодействуют с модулями, с которыми мы встречались ранее в главах 3 и 4:

- *launchmodes.py* запускает независимые программы Python переносимым образом.
- *Launcher.py* находит программы и в конечном итоге запускает PyDemos и PyGadgets при использовании самоконфигурирующимися сценариями верхнего уровня для запуска.
- *LaunchBrowser.py* запускает веб-броузеры.

Код этих модулей находится в примерах 3.24, 4.14 и 4.15. Перечисленные здесь программы добавляют в систему запуска программ компоненты GUI – они создают простые в использовании кнопки, при нажатии которых запускается большая часть крупных примеров, содержащихся в тексте.

Оба эти сценария предполагают, что при их запуске рабочим был каталог, в котором они содержатся (в них жестко прошиты пути к другим программам относительно этого каталога). Щелкните по их именам в менеджере файлов или запустите из команд-

[1] Origin Systems, крупная компания по производству игровых программ, использует в таком качестве Python для управления анимацией в некоторых из своих игр. В последних сообщениях говорилось, что их продукт для сетевой игры Ultima Online II должен был управляться Python.

ной строки после выполнения *cd* в корневой каталог примеров *PP2E*. Эти сценарии можно было бы сделать запускаемыми из других каталогов путем использования значения переменной окружения `PP2EHOME` для получения путей к сценариям, но в действительности они предназначены только для запуска из корневого каталога *PP2E*.

Панель запуска PyDemos

Сценарий PyDemos строит панель кнопок, которые запускают программы в демонстрационном режиме – не для повседневного применения. Я использую PyDemos для показа программ Python при каждой возможности – значительно легче нажимать на кнопки, чем набирать командные строки или искать сценарии через GUI файлового менеджера. Следует пользоваться PyDemos для запуска и работы с примерами, представленными в этой книге – все кнопки этого GUI представляют примеры, с которыми мы познакомимся в последующих главах.

Чтобы пользоваться этой панелью запуска было еще легче, перетащите ее на рабочий стол Windows, создав ярлык, по которому можно щелкнуть мышью (на других системах можно проделать нечто аналогичное). Так как в этом сценарии жестко запрограммированы командные строки для запуска программ, находящихся в других местах дерева примеров, он также полезен как указатель к главным примерам из книги. На рис. 8.39 показано, как выглядит PyDemos при выполнении в Windows; в Linux он выглядит несколько иначе, но действует так же.

Рис. 8.39. PyDemos со своими всплывающими окнами

Исходный код, с помощью которого создается такая картина, приведен в примере 8.32. PyDemos не представляет чего-либо особенного в отношении программирования GUI. Его функция `demoButton` просто прикрепляет к главному окну новую кнопку, готовую при нажатии породить программу Python. Для запуска программ PyDemos вызывает экземпляр объекта `launchmodes.PortableLauncher`, с которым мы познакомились в конце главы 3, – поскольку он выступает здесь в роли обработчика обратного вызова Tkinter, для запуска программы используется операция *call*.

Как показано на рис. 8.39, PyDemos строит также два всплывающих окна, когда нажимаются кнопки внизу главного окна – окно Info с кратким описанием последней запущенной демонстрационной программы и окно Links с переключателями, нажатие которых открывает связанные с книгой сайты в локальном веб-броузере:

- Всплывающее окно Info выводит простую строку сообщения и раз в секунду изменяет ее шрифт, чтобы привлечь к себе внимание. Поскольку это может раздражать, всплывающее окно сначала появляется в свернутом виде (щелкните по кнопке Info, чтобы посмотреть на него или спрятать).
- Переключатели всплывающего окна Links похожи на гиперссылки на веб-странице, но этот GUI – не броузер: при нажатии на них с помощью сценария `LaunchBrowser`, с которым мы познакомились в предыдущей главе, отыскивается и запускается веб-броузер, подключающийся к соответствующему сайту при наличии соединения с Интернетом.

PyDemos работает в Windows и в Linux в основном благодаря внутренне присущей переносимости как Python, так и Tkinter. Дополнительные подробности можно найти в исходном коде, представленном в примере 8.32.

Пример 8.32. PP2E\PyDemos.pyw

```
########################################################################
# PyDemos.pyw
# Programming Python, 2nd Edition (PP2E), 1999--2001
#
# Запуск основных примеров GUI Python+Tk из этой книги способом, независимым от платформы.
# Этот файл служит также указателем к основным примерам программ, хотя многие примеры не имеют
# GUI и потому здесь не перечислены (см., например, в сценариях компиляции gcc для Linux
# в корневом каталоге примеров указатели программ интеграции C). См. также:
#
# - PyGadgets.py, более простой сценарий для запуска программ
#   в недемонстрационном режиме, если нужно использовать их обычным образом
# - PyGadgets_bar.pyw, который создает панель кнопок для запуска всех
#   программ PyGadgets по отдельности, а не всех сразу
# - Launcher.py для запуска программ без настройки окружения -
#   находит Python, устанавливает PYTHONPATH и т. д.
# - Launch_*.py для запуска PyDemos и PyGadgets с помощью
#   Launcher.py – запускайте для беглого знакомства
# - LaunchBrowser.py для открытия веб-страниц примеров
#   в автоматически обнаруживаемом веб-броузере
# - README-PP2E.txt, для общей информации о примерах
#
# Использующие Интернет демонстрационные программы лежат здесь:
#     http://starship.python.net/~lutz/PyInternetDemos.html
# но эта программа пытается автоматически запустить броузер с главными веб-страницами,
# находящимися на указанном сайте или в локальных файлах. Дополнительные комментарии
# к программам были перемещены в файл PyDemos.doc.txt
########################################################################

import sys, time, os, launchmodes
from Tkinter import *

# -live загружает корневые страницы из сети, -file загружает локальные файлы
InternetMode = '-file'

######################################
# начать создание главных окон gui
######################################

Root = Tk()
Root.title('PP2E Demos')
```

```
# построить окно сообщений
Stat = Toplevel()
Stat.protocol('WM_DELETE_WINDOW', lambda:0)    # игнорировать wm delete
Stat.title('PP2E demo info')

Info = Label(Stat, text = 'Select demo',
             font=('courier', 20, 'italic'), padx=12, pady=12, bg='lightblue')
Info.pack(expand=YES, fill=BOTH)

############################################################
# добавить кнопки запуска с объектами обратного вызова
############################################################

# класс запуска демо
class Launcher(launchmodes.PortableLauncher):    # использовать класс-оболочку
    def announce(self, text):                    # настройка для установки метки в GUI
        Info.config(text=text)

def demoButton(name, what, where):
    b = Button(Root, bg='navy', fg='white', relief=RIDGE, border=4)
    b.config(text=name, command=Launcher(what, where))
    b.pack(side=TOP, expand=YES, fill=BOTH)

demoButton('PyEdit',
           'Text file editor',                                  # редактировать
           'Gui.TextEditor.textEditor.pyw PyDemos.pyw')     # предполагается в текущем
                                                                # рабочем каталоге
demoButton('PyView',
           'Image slideshow, plus note editor',
           'Gui/SlideShow/slideShowPlus.py Gui/gifs')
demoButton('PyDraw',
           'Draw and move graphics objects',
           'Gui/MovingPics/movingpics.py Gui/gifs')
demoButton('PyTree',
           'Tree data structure viewer',
           'Dstruct/TreeView/treeview.py')
demoButton('PyClock',
           'Analog/digital clocks',
           'Gui/Clock/clockStyles.py Gui/gifs')
demoButton('PyToe',
           'Tic-tac-toe game (AI)',
           'Ai/TicTacToe/tictactoe.py')
demoButton('PyForm',                                    # просмотр словаря в памяти
           'Persistent table viewer/editor',            # или сменить текущий каталог
                                                        # на каталог с файлами shelve для класса
          #'Dbase/TableBrowser/formgui.py')             # 0=не реинициализировать shelve
          #'Dbase/TableBrowser/formtable.py  shelve 0 pyformData-1.5.2')
           'Dbase/TableBrowser/formtable.py  shelve 1 pyformData')
demoButton('PyCalc',
           'Calculator, plus extensions',
           'Lang/Calculator/calculator_plusplus.py')
demoButton('PyMail',
           'Python+Tk pop/smtp email client',
           'Internet/Email/PyMailGui.py')
demoButton('PyFtp',
           'Python+Tk ftp clients',
           'Internet/Ftp/PyFtpGui.pyw')

if InternetMode == '-file':
    pagepath = os.getcwd() + '/Internet/Cgi-Web'
```

```
    demoButton('PyErrata',
               'Internet-based errata report system',
               'LaunchBrowser.py -file %s/PyErrata/pyerrata.html' % pagepath)
    demoButton('PyMailCgi',
               'Browser-based pop/smtp email interface',
               'LaunchBrowser.py -file %s/PyMailCgi/pymailcgi.html' % pagepath)
    demoButton('PyInternet',
               'Internet-based demo launcher page',
               'LaunchBrowser.py -file %s/PyInternetDemos.html' % pagepath)
else:
    site = 'starship.python.net/~lutz'
    demoButton('PyErrata',
               'Internet-based errata report system',
               'LaunchBrowser.py -live PyErrata/pyerrata.html ' + site)
    demoButton('PyMailCgi',
               'Browser-based pop/smtp email interface',
               'LaunchBrowser.py -live PyMailCgi/pymailcgi.html ' + site)
    demoButton('PyInternet',
               'Main Internet demos launcher page',
               'LaunchBrowser.py -live PyInternetDemos.html ' + site)

#Попробовать: привязать событие появления мыши к изменению текста
#См. также: site http://starship.python.net/~lutz/PyInternetDemos.html

############################################################
# переключение шрифта в окне сообщения раз в секунду
############################################################

def refreshMe(info, ncall):
    slant = ['normal', 'italic', 'bold', 'bold italic'][ncall % 4]
    info.config(font=('courier', 20, slant))
    Root.after(1000, (lambda info=info, ncall=ncall: refreshMe(info, ncall+1)) )

############################################################
# показать/спрятать окно статуса при щелчке по info
############################################################

Stat.iconify()
def onInfo():
    if Stat.state() == 'iconic':
        Stat.deiconify()
    else:
        Stat.iconify()  # was 'normal'

################################################################
# вывести ряд кнопок со ссылками веб, если есть соединение
################################################################

radiovar = StringVar() # use a global

def onLinks():
    popup = Toplevel()
    popup.title('PP2E web site links')
    links = [("Book",     'LaunchBrowser.py -live about-pp.html rmi.net/~lutz'),
             ("Python",   'LaunchBrowser.py -live index.html www.python.org'),
             ("O'Reilly", 'LaunchBrowser.py -live index.html www.oreilly.com'),
             ("Author",   'LaunchBrowser.py -live index.html rmi.net/~lutz')]

    for (name, command) in links:
        callback = Launcher((name + "'s web site"), command)
        link = Radiobutton(popup, text=name, command=callback)
```

```
        link.config(relief=GROOVE, variable=radiovar, value=name)
        link.pack(side=LEFT, expand=YES, fill=BOTH)
    Button(popup, text='Quit', command=popup.destroy).pack(expand=YES,fill=BOTH)

    if InternetMode != '-live':
        from tkMessageBox import showwarning
        showwarning('PP2E Demos', 'Web links require an Internet connection')

############################################################
# завершение создания основного gui, начало цикла событий
############################################################

Button(Root, text='Info',  command=onInfo).pack(side=TOP, fill=X)
Button(Root, text='Links', command=onLinks).pack(side=TOP, fill=X)
Button(Root, text='Quit',  command=Root.quit).pack(side=BOTTOM, fill=X)
refreshMe(Info, 0)  # start toggling
Root.mainloop()
```

Панель запуска PyGadgets

Сценарий PyGadgets запускает часть тех же программ, что и PyDemos, но для реального практического использования, а не как кратковременные демонстрации. Оба сценария запускают другие программы и выводят панель с кнопками с помощью `launchmodes`, но этот немного проще, потому что его задача более узкая. PyGadgets также поддерживает два режима вызова: он может сразу запустить одновременно все программы из готового списка или вывести GUI для запуска каждой программы отдельно (рис.8.40 показывает GUI панели кнопок при запуске по требованию).

Из-за этих различий подход к построению GUI в PyGadgets в большей мере основывается на данных: имена программ хранятся в списке, который просматривается при необходимости, вместо использования ряда готовых вызовов `demoButton`. Например, набор кнопок в GUI панели запуска на рис. 8.40 целиком зависит от содержимого списка программ.

Рис. 8.40 Панель запуска PyGadgets

Исходный код этого GUI приведен в примере 8.33. Он невелик по объему, потому что основывается на других модулях (`launchmodes`, `LaunchBrowser`), осуществляющих большую часть его действий. На моих машинах PyGadgets всегда открыт (на рабочем столе Windows у меня также есть ярлык этого сценария). С его помощью я легко получаю доступ к повседневно используемым инструментам – текстовым редакторам, калькуляторам и так далее, – которые все встретятся нам в будущих главах.

Для настройки PyGadgets под собственные нужды просто импортируйте и вызывайте его функции через свои списки командных строк программ или измените список `mytools` вызываемых программ, который находится ближе к концу файла. В конце концов, это Python.

Пример 8.33. PP2E\PyGadgets.py

```
#!/bin/env python
#########################################################################################
# Запуск разных примеров; чтобы сделать их постоянно доступными, запускайте сценарий
# при загрузке системы. Файл предназначен для запуска программ, действительно необходимых
```

```
# в работе; для запуска демонстрационных программ Python/Tk и получения дополнительных
# сведений о параметрах запуска программ обратитесь к PyDemos.
# Замечание о работе под Windows: это файл с расширением '.py', поэтому при щелчке по нему
# выводится окно консоли dos; окно dos используется для вывода начального сообщения
# (с засыпанием на 5 секунд, чтобы его было видно, пока запускаются приложения).
# Если не хотите вывода окна dos, запускайте сценарий через программу 'pythonw'
# (а не 'python'), используйте расширение '.pyw', пометьте свойством Windows 'run minimized'
# или порождайте файл из другого места; см. PyDemos.
##########################################################################################

import sys, time, os, time
from Tkinter import *
from launchmodes import PortableLauncher           # повторное использование класса запуска
программ

def runImmediate(mytools):
    # запустить приложения немедленно
    print 'Starting Python/Tk gadgets...'          # сообщения во временный экран stdout
    for (name, commandLine) in mytools:
        PortableLauncher(name, commandLine)()      # сразу вызвать для запуска
    print 'One moment please...'                   # \b означает "забой"
    if sys.platform[:3] == 'win':
        # в Windows сохранять окно консоли stdio 5 секунд
        for i in range(5): time.sleep(1); print ('\b' + '.'*10),

def runLauncher(mytools):
    # организовать простую панель запуска для использования в дальнейшем
    root = Tk()
    root.title('PyGadgets PP2E')
    for (name, commandLine) in mytools:
        b = Button(root, text=name, fg='black', bg='beige', border=2,
                   command=PortableLauncher(name, commandLine))
        b.pack(side=LEFT, expand=YES, fill=BOTH)
    root.mainloop()

mytools = [
    ('PyEdit',   'Gui/TextEditor/textEditor.pyw'),
    ('PyView',   'Gui/SlideShow/slideShowPlus.py Gui/gifs'),
    ('PyCalc',   'Lang/Calculator/calculator.py'),
    ('PyMail',   'Internet/Email/PyMailGui.py'),
    ('PyClock',  'Gui/Clock/clock.py -size 175 -bg white'
                 ' -picture Gui/gifs/pythonPowered.gif'),
    ('PyToe',    'Ai/TicTacToe/tictactoe.py'
                 ' -mode Minimax -fg white -bg navy'),
    ('PyNet',    'LaunchBrowser.py -file ' + os.getcwd() +
                 '/Internet/Cgi-Web/PyInternetDemos.html')
]

if __name__ == '__main__':
    prestart, toolbar = 1, 0
    if prestart:
        runImmediate(mytools)
    if toolbar:
        runLauncher(mytools)
```

По умолчанию PyGadgets сразу запускает программы. Для выполнения PyGadgets в режиме панели запуска в примере 8.34 импортируется и вызывается соответствующая функция с импортированным списком программ. Так как это файл типа *.pyw*, видно только изначально создаваемый GUI панели запуска, а не окно консоли DOS.

Пример 8.34. PP2E\PyGadgets_bar.pyw

```
# запуск только панели PyGadgets вместо одновременного запуска всех
# приложений; имя файла предотвращает появление окна dos в Windows

import PyGadgets
PyGadgets.runLauncher(PyGadgets.mytools)
```

Этот сценарий – тот файл, который вызывается ярлыком на моем рабочем столе: я предпочитаю GUI вызова приложений по требованию. Такой сценарий можно выполнять и при загрузке системы, чтобы сделать его постоянно доступным (и сэкономить на щелчке мышью). Например:

- В Windows такой сценарий автоматически запускается при добавлении его в папку Автозагрузка – нажмите кнопку Пуск, выберите Настройка, перейдите в диалог Панель задач и меню Пуск и проделайте оставшиеся шаги.
- В Linux и Unix можно автоматически запускать этот сценарий из командной строки в стартовых сценариях (например, *.cshrc, .profile* или *.login* в исходном каталоге) после запуска X Windows.

Каким бы способом ни был запущен PyGadgets – через ярлык, щелчком в менеджере файлов, через командную строку или иным образом, – появляется панель запуска, показанная вверху рис. 8.41.

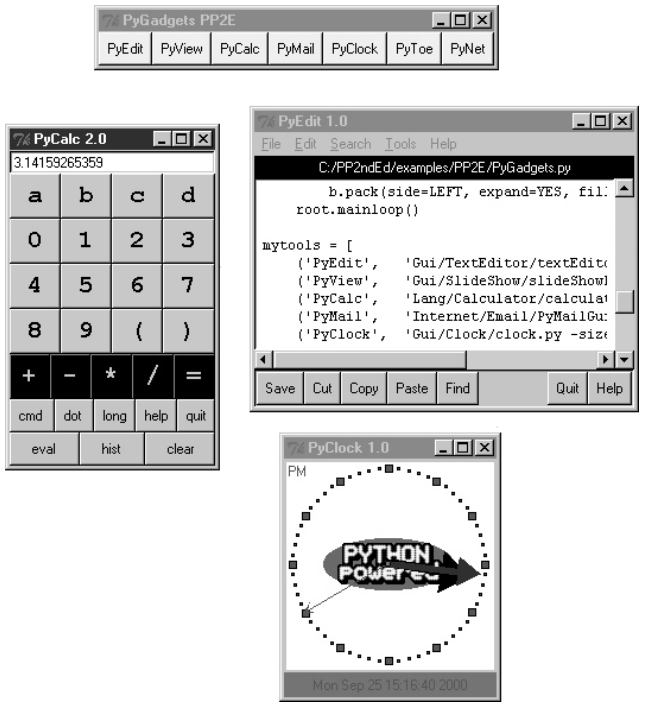


Рис. 8.41. Панель запуска PyGadgets

Конечно, весь смысл PyGadgets состоит в том, чтобы запускать другие программы. При нажатии на кнопки панели запуска запускаются программы, показанные в нижней части рис. 8.41, и если вы хотите узнать о них больше, придется перевернуть страницу и перейти к следующей главе.

9

Более крупные примеры GUI

«Как улучшить мышеловку»

В этой главе мы продолжаем изучать создание графических интерфейсов пользователей с помощью Python и его стандартной библиотеки Tkinter путем представления ряда реальных программ GUI. В трех предшествующих главах мы познакомились с основами программирования с Tkinter и обследовали базовый набор графических элементов (*widgets*) – классов Python, которые генерируют инструменты на экране компьютера и могут реагировать на события, вызываемые пользователем, например щелчки мышью. В данной главе мы займемся тем, что будем создавать более полезные GUI, соединяя эти графические элементы вместе. Нами будут изучены:

- Развитые технологии кодирования GUI
- *PyEdit* – программа текстового редактора
- *PyView* – программа слайд-шоу графических изображений
- *PyDraw* – графический редактор
- *PyClock* – графические часы
- *PyToe* – и даже простая игра в качестве развлечения[1]

Как и в главе 4 «Более крупные системные примеры, часть 1» и в главе 5 «Более крупные системные примеры, часть 2», я выбрал примеры этой главы из собственной библиотеки программ Python, которыми я действительно пользуюсь. Например, GUI текстового редактора и часов, с которыми мы здесь познакомимся, служат рабочими лошадками, изо дня в день используемыми мной на моих машинах. Так как они написаны на Python и Tkinter, то без изменений работают на машинах Windows и Linux и должны работать также на Mac.

А так как это сценарии на чистом Python, их дальнейшее развитие целиком зависит от пользователей – освоившись с интерфейсами Tkinter, изменить или улучшить поведение таких программ редактированием их кода Python легко. Хотя некоторые из этих примеров аналогичны коммерческим программам (например, PyEdit напоминает Windows Notepad), переносимость и почти безграничная настраиваемость сценариев Python дают явное преимущество.

---

[1] У всех более крупных примеров этой книги в начале имени стоит «Py». Это соглашение, принятое в мире Python. Если порыться на *http://www.python.org*, то можно найти другое свободно распространяемое программное обеспечение, следующее этой схеме: PyApache (интерфейс Python к веб-серверу Apache), PySol (игра в солитер на Python/Tkinter) и многие другие. Я не знаю, с кого началась эта схема, но она оказалась достаточно тонким способом рекламы языка программирования для всего мира программного обеспечения с открытым кодом. Если Питонист слишком прямолинеен – это не Питонист!

Примеры в других главах

Далее в этой книге мы встретим другие программы GUI на базе Tkinter, представляющие удобные интерфейсы для конкретных областей приложений. В частности, в последующих главах появятся такие более крупные примеры GUI:

- *PyMail* – клиент электронной почты в главе 11 «Создание сценариев, выполняемых на стороне клиента»
- *PyForm* – средство просмотра таблиц постоянных объектов в главе 16 «Базы данных и постоянное хранение данных»
- *PyTree* – средство просмотра древовидных структур данных в главе 17 «Структуры данных»
- *PyCalc* – калькулятор в главе 18 «Текст и язык»

Большинством из этих программ я постоянно пользуюсь. Так как библиотеки GUI являются инструментами общего назначения, не много найдется областей приложений, которые не выиграли бы от простого в использовании, простого в программировании и хорошо переносимого интерфейса, код которого написан на Python с Tkinter.

Помимо примеров, приведенных в этой книге, для Python существуют инструментальные наборы GUI более высокого уровня, например система PMW, упомянутая в главе 6 «Графические интерфейсы пользователя». Такие системы, основываясь на Tkinter, предоставляют составные компоненты, например блокнот или элементы с закладками. Позднее мы встретимся также с программами, которые строят интерфейсы пользователя в веб-броузерах, а не Tkinter. Но за исключением простых, основанных на веб интерфейсов, GUI на базе Tkinter могут оказаться необходимой характеристикой почти каждой создаваемой на Python программы.

Стратегия данной главы

Как и все главы этой книги, посвященные исследованию конкретного случая, данная глава в значительной мере является «обучением на примере»; текст большинства программ приведен с минимумом подробностей. Попутно я буду отмечать новые функции Tkinter, появляющиеся в каждом примере, но я также буду предполагать, что вы самостоятельно изучите детали по приведенному исходному коду и комментариям. Легкость чтения Python становится существенным достоинством для программистов (и авторов), особенно когда сложность программ достигает такого уровня, как в этой главе.

Наконец, я хочу напомнить, что все перечисленные выше крупные программы можно запускать из GUI панелей запуска PyDemos и PyGadgets, с которыми мы встретились в конце прошлой главы. Я попытаюсь передать их поведение на снимках экранов, которые будут приведены, но GUI по своей природе являются системами, управляемыми событиями, и реальный запуск их для того, чтобы почувствовать характер взаимодействия с пользователем, ничем не заменишь. Поэтому панели запуска являются фактическим дополнением к материалу данной главы. Они могут выполняться на большинстве платформ и обеспечивают легкость запуска (см. файл *README-PP2E.txt*). Запускайте их и сразу начинайте щелкать мышью, если еще не сделали этого.

Более сложные приемы написания кода GUI

Если вы прочли главу 8 «Обзор Tkinter, часть 2», то знаете, что код, строящий нетривиальные GUI, может достичь большого размера, если каждый графический элемент строить вручную. Приходится не только вручную связывать все графические элемен-

ты, но нужно помнить десятки параметров, которые должны быть установлены. При такой стратегии программирование GUI часто становится упражнением по вводу с клавиатуры или операциям вырезания и вставки в текстовом редакторе.

GuiMixin: использование общих методов «подмешиваемых» классов

Вместо выполнения всех действий вручную правильно было бы создать оболочку или автоматизировать в возможно большей степени процесс построения GUI. Одним из решений является создание функций, обеспечивающих типичные конфигурации графических элементов; например можно было бы определить функцию кнопки, работающую с деталями настройки и поддерживающую большинство рисуемых кнопок.

Другим способом может быть реализация общих методов в классе и наследование их в необходимых случаях. Такие классы обычно называют *подмешиваемыми* (*mixin*), потому что их методы «подмешиваются» в другие классы. Такие классы оформляют полезные во многих случаях инструменты в виде методов. Идея близка к импортированию модулей, однако подмешиваемые классы могут обращаться к конкретному экземпляру, *self*, используя состояние конкретного объекта и унаследованные методы. Пример 9.1 демонстрирует, как это делается.

Пример 9.1. PP2E\Gui\Tools\guimixin.py

```
##############################################################################################
# "подмешиваемый" класс для других фреймов: общие методы для готовых диалогов, порождения
# программ и т. д.; должен смешиваться с классом, производным от Frame, ради его метода quit
##############################################################################################

from Tkinter import *
from tkMessageBox import *
from tkFileDialog import *
from ScrolledText import ScrolledText
from PP2E.launchmodes import PortableLauncher, System

class GuiMixin:
    def infobox(self, title, text, *args):              # использовать стандартные диалоги
        return showinfo(title, text)                    # *args для обратной совместимости
    def errorbox(self, text):
        showerror('Error!', text)
    def question(self, title, text, *args):
        return askyesno(title, text)

    def notdone(self):
        showerror('Not implemented', 'Option not available')
    def quit(self):
        ans = self.question('Verify quit', 'Are you sure you want to quit?')
        if ans == 1:
            Frame.quit(self)                            # quit не рекурсивен!
    def help(self):
        self.infobox('RTFM', 'See figure 1...')         # переопределите его более подходящим

    def selectOpenFile(self, file="", dir="."):         # использовать стандартные диалоги
        return askopenfilename(initialdir=dir, initialfile=file)
    def selectSaveFile(self, file="", dir="."):
        return asksaveasfilename(initialfile=file, initialdir=dir)

    def clone(self):
        new = Toplevel()                    # создать новую версию
```

```
        myclass = self.__class__          # объект экземпляра (самого низшего) класса
        myclass(new)                      # прикрепить/выполнить экземпляр к новому окну

    def spawn(self, pycmdline, wait=0):
        if not wait:
            PortableLauncher(pycmdline, pycmdline)()     # запустить программу Python
        else:
            System(pycmdline, pycmdline)()               # ждать ее завершения

    def browser(self, filename):
        new  = Toplevel()                                # создать новое окно
        text = ScrolledText(new, height=30, width=90)    # Text с полосой прокрутки
        text.config(font=('courier', 10, 'normal'))      # моноширинный шрифт
        text.pack()
        new.title("Text Viewer")                         # атрибуты менеджера окон
        new.iconname("browser")
        text.insert('0.0', open(filename, 'r').read() )  # вставить текст файла

if __name__ == '__main__':
    class TestMixin(GuiMixin, Frame):      # автономное тестирование
        def __init__(self, parent=None):
            Frame.__init__(self, parent)
            self.pack()
            Button(self, text='quit',  command=self.quit).pack(fill=X)
            Button(self, text='help',  command=self.help).pack(fill=X)
            Button(self, text='clone', command=self.clone).pack(fill=X)
    TestMixin().mainloop()
```

Хотя пример 9.1 ориентирован на GUI, в действительности он иллюстрирует идеи архитектуры. Класс `GuiMixin` реализует частые операции со стандартными интерфейсами, которые безразличны к возможным изменениям в реализации. На самом деле реализации некоторых методов этого класса действительно изменились – в промежутке между первым и вторым изданиями этой книги вызовы `Dialog` в старом стиле были заменены новыми вызовами стандартных диалогов Tk. Так как интерфейс этого класса скрывает такие детали, не потребовалось изменять его клиенты, чтобы они использовали новую технологию диалогов.

В данном виде `GuiMixin` предоставляет методы для стандартных диалогов, клонирования окон, порождения программ, просмотра текстовых файлов и т. д. Позднее, если обнаружится, что одни и те же методы приходится кодировать многократно, их можно добавить в такой подмешиваемый класс, и они станут немедленно доступны всюду, куда он импортируется и внедряется. Более того, методы `GuiMixin` можно наследовать и использовать в существующем виде либо переопределять в подклассе.

Здесь нужно отметить несколько вещей:

- Метод `quit` выполняет отчасти ту же задачу, что и кнопка многократного использования `Quitter` в предыдущих главах. Так как в подмешиваемых классах могут определяться большие библиотеки многократно используемых методов, они могут предоставлять более мощный способ упаковки многократно используемых компонентов, чем отдельные классы. Если правильно упаковать подмешиваемый класс, то он может дать значительно больше, чем обратный вызов одной кнопки.
- Метод `clone` создает новый экземпляр самого узкого класса, который смешивается с `GuiMixin`, в новом окне верхнего уровня (`self.__class__` является объектом класса, из которого был создан экземпляр). Он открывает новый независимый экземпляр окна.
- Метод `browser` открывает в новом окне объект стандартной библиотеки `ScrolledText` и заполняет его текстом файла, который нужно просмотреть. В предыдущей главе

мы написали собственный ScrolledText, который может потребоваться в данном случае, если класс стандартной библиотеки когда-либо будет объявлен устаревшим (но не стоит делать на это ставки).

- Метод spawn запускает командную строку программы Python как новый процесс и либо ждет его завершения, либо нет (в зависимости от аргумента wait). Этот метод прост потому, что детали запуска скрыты в модуле launchmodes, показанном в конце главы 3 «Системные средства параллельного выполнения». GuiMixin поощряет и использует на практике хорошие привычки повторного использования кода.

Класс GuiMixin предназначен служить библиотекой многократно используемых инструментальных методов и сам по себе, в сущности, бесполезен. В действительности для использования его нужно смешивать с классом, основанным на Frame: quit предполагает, что он смешивается с Frame, а clone предполагает, что он смешивается с классом графического элемента. Чтобы удовлетворить этим ограничениям, находящийся в конце код для самотестирования объединяет GuiMixin с графическим элементом Frame. На рис. 9.1 показана картина, которая возникает при самотестировании после того, как дважды нажата кнопка «clone» и затем «help» в одном из трех экземпляров.

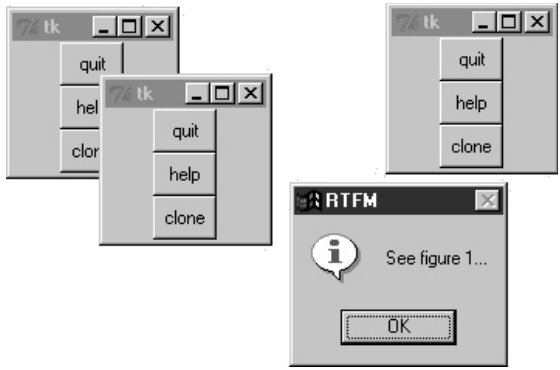


Рис. 9.1. Код самотестирования GuiMixin в действии

Мы снова встретимся с использованием этого класса в качестве mixin в последующих примерах – в конце концов, в этом весь смысл повторного использования кода.

GuiMaker: автоматизация создания меню и панелей инструментов

Подмешиваемый класс из последнего раздела облегчает выполнение стандартных задач, но не решает проблем сложности связывания в таких графических элементах, как меню и панели инструментов. Конечно, при наличии инструмента структурирования GUI, который генерировал бы код Python, проблем бы не было. Мы бы проектировали графические элементы интерактивно, нажимали кнопку и заполняли пустые места в обработчике обратного вызова.

Но пока может сгодиться и подход, основанный на программировании. Хотелось бы иметь возможность наследовать некоторый класс, который сам выполняет всю черновую работу, если задать шаблон для меню и панелей инструментов в окне. Вот один из способов, которыми это можно сделать, использующий деревья простых объектов. Класс примера 9.2 интерпретирует представление меню и панелей инструментов в виде структуры данных и автоматически строит все графические элементы.

Пример 9.2. PP2E\Gui\Tools\guimaker.py

```
##############################################################################################
# Расширенный Frame, автоматически создающий оконные меню и панели инструментов.
# GuiMakerFrameMenu предназначен для встроенных компонентов (делает меню на основе фреймов).
# GuiMakerWindowMenu предназначен для окон верхнего уровня (делает оконные меню Tk8.0).
# Пример формата дерева структуры см. в коде самотестирования (и PyEdit).
##############################################################################################

import sys
from Tkinter import *                           # классы графических элементов
from types   import *                           # константы типов

class GuiMaker(Frame):
    menuBar    = []                             # значения по умолчанию; изменять
    toolBar    = []                             # при создании подклассов
    helpButton = 1                              # устанавливать в start(), если нужен self

    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)        # растягиваемый фрейм
        self.start()                            # в подклассе: установить menu/toolBar
        self.makeMenuBar()                      # панель меню строится здесь
        self.makeToolBar()                      # панель инструментов строится здесь
        self.makeWidgets()                      # в подклассе: добавить середину

    def makeMenuBar(self):
        """
        сделать панель меню вверху (меню Tk8.0 внизу)
        expand=no, fill=x, чтобы ширина не менялась
        """
        menubar = Frame(self, relief=RAISED, bd=2)
        menubar.pack(side=TOP, fill=X)

        for (name, key, items) in self.menuBar:
            mbutton  = Menubutton(menubar, text=name, underline=key)
            mbutton.pack(side=LEFT)
            pulldown = Menu(mbutton)
            self.addMenuItems(pulldown, items)
            mbutton.config(menu=pulldown)

        if self.helpButton:
            Button(menubar, text    = 'Help',
                            cursor  = 'gumby',
                            relief  = FLAT,
                            command = self.help).pack(side=RIGHT)

    def addMenuItems(self, menu, items):
        for item in items:                      # просмотр списка вложенных элементов
            if item == 'separator':             # строка: добавить разделитель
                menu.add_separator({})
            elif type(item) == ListType:        # список: отключенных элементов
                for num in item:
                    menu.entryconfig(num, state=DISABLED)
            elif type(item[2]) != ListType:
                menu.add_command(label     = item[0],           # команда:
                                 underline = item[1],           # добавить команду
                                 command   = item[2])           # что вызвать
            else:
                pullover = Menu(menu)
                self.addMenuItems(pullover, item[2])            # подсписок:
```

```
                menu.add_cascade(label     = item[0],          # создать подменю
                                 underline = item[1],          # добавить каскад
                                 menu      = pullover)

    def makeToolBar(self):
        """
        если нужно, сделать внизу панель кнопок
        expand=no, fill=x, чтобы ширина не изменялась
        """
        if self.toolBar:
            toolbar = Frame(self, cursor='hand2', relief=SUNKEN, bd=2)
            toolbar.pack(side=BOTTOM, fill=X)
            for (name, action, where) in self.toolBar:
                Button(toolbar, text=name, command=action).pack(where)

    def makeWidgets(self):
        """
        'средняя' часть создается последней, чтобы меню/панель инструментов
        всегда были вверху/внизу и обрезались последними;
        переопределяйте: упаковывайте середину к любому краю;
        для grid: располагайте среднюю часть в упаковываемом фрейме
        """
        name = Label(self,
                     width=40, height=10,
                     relief=SUNKEN, bg='white',
                     text   = self.__class__.__name__,
                     cursor = 'crosshair')
        name.pack(expand=YES, fill=BOTH, side=TOP)

    def help(self):
        """
        переопределить в подклассе
        """
        from tkMessageBox import showinfo
        showinfo('Help', 'Sorry, no help for ' + self.__class__.__name__)

    def start(self): pass  # переопределить в подклассе

################################################################################
# Для панели меню главного окна Tk 8.0 вместо фрейма
################################################################################

GuiMakerFrameMenu = GuiMaker                  # используется для меню встраиваемых компонентов

class GuiMakerWindowMenu(GuiMaker):           # используется для меню окон верхнего уровня
    def makeMenuBar(self):
        menubar = Menu(self.master)
        self.master.config(menu=menubar)

        for (name, key, items) in self.menuBar:
            pulldown = Menu(menubar)
            self.addMenuItems(pulldown, items)
            menubar.add_cascade(label=name, underline=key, menu=pulldown)

        if self.helpButton:
            if sys.platform[:3] == 'win':
                menubar.add_command(label='Help', command=self.help)
            else:
                pulldown = Menu(menubar)      # для linux необходимо настоящее ниспадающее меню
                pulldown.add_command(label='About', command=self.help)
                menubar.add_cascade(label='Help', menu=pulldown)
```

```
##############################################################################
# Самотестирование при автономном выполнении: 'python guimaker.py'
##############################################################################

if __name__ == '__main__':
    from guimixin import GuiMixin            # встроить метод help

    menuBar = [
        ('File', 0,
            [('Open',  0, lambda:0),         # lambda:0 - пустая операция
             ('Quit',  0, sys.exit)]),       # здесь sys, а не self
        ('Edit', 0,
            [('Cut',   0, lambda:0),
             ('Paste', 0, lambda:0)]) ]
    toolBar = [('Quit', sys.exit, {'side': LEFT})]

    class TestAppFrameMenu(GuiMixin, GuiMakerFrameMenu):
        def start(self):
            self.menuBar = menuBar
            self.toolBar = toolBar
    class TestAppWindowMenu(GuiMixin, GuiMakerWindowMenu):
        def start(self):
            self.menuBar = menuBar
            self.toolBar = toolBar
    class TestAppWindowMenuBasic(GuiMakerWindowMenu):
        def start(self):
            self.menuBar = menuBar
            self.toolBar = toolBar            # help из guimaker, а не из guimixin

    root = Tk()
    TestAppFrameMenu(Toplevel())
    TestAppWindowMenu(Toplevel())
    TestAppWindowMenuBasic(root)
    root.mainloop()
```

Чтобы понять этот модуль, нужно быть знакомым с основами создания меню, изложенными в предыдущей главе. Тогда код ясен: класс `GuiMaker` обходит структуры меню и панели инструментов и попутно создает соответствующие графические элементы. В код самотестирования включен простой пример структур данных, описывающих меню и панели инструментов:

Шаблоны меню

Списки и вложенные подсписки троек (*метка, подчеркивание_для_быстрого_вызова, обработчик*). Если `обработчик` является подсписком, а не функцией или методом, предполагается, что это каскадное подменю.

Шаблоны панелей инструментов

Список троек (*метка, обработчик, параметры_упаковки*). Параметры упаковки записываются как словарь параметров, передаваемых методу графического элемента `pack` (он принимает словари, но можно преобразовать словарь в аргументы – ключевые слова, передав его в качестве третьего аргумента в `apply`).

Протоколы подклассов

Помимо структур меню и панелей инструментов клиенты этого класса могут вмешиваться и изменять реализованные в нем протоколы методов и геометрии:

Атрибуты шаблона

Предполагается, что клиенты этого класса устанавливают атрибуты `menuBar` и `toolBar` в каком-то месте цепочки наследования к моменту завершения метода `start`.

Инициализация

Метод `start` может переопределяться для динамического создания шаблонов меню и панели инструментов (поскольку будет доступен `self`); `start` служит также местом осуществления общей инициализации – конструктор `__init__` класса `GuiMixin` должен быть выполнен, а не переопределен.

Добавление графических элементов

Метод `makeWidgets` может быть переопределен так, чтобы создавать среднюю часть окна – место в приложении между панелями меню и инструментов. По умолчанию `makeWidgets` помещает в середине метку с именем самого узкого класса, но по сути это абстрактный метод и предполагается его конкретизация.

Протокол упаковки

В конкретном методе `makeWidgets` клиенты могут прикреплять графические элементы из средней части к любому краю «self» (`Frame`), так как панели меню и инструментов уже захватили верх и низ контейнера к моменту выполнения `makeWidgets`. Если составляющие среднюю часть элементы упаковываются, она не обязательно должна быть вложенным фреймом. Панели меню и инструментов автоматически упаковываются первыми, чтобы быть обрезанными в последнюю очередь при сжатии окна.

Протокол сетки

В средней части расположение элементов может осуществляться по сетке, если эта сетка помещена во вложенный `Frame`, который упаковывается в родительский `self`. (Помните, что на каждом уровне контейнеров можно применять `grid` или `pack`, но не оба вместе, а `self` является `Frame` с уже упакованными панелями ко времени вызова `makeWidgets`.) Так как `Frame GuiMaker` упаковывает себя в родительском контейнере, по аналогичным причинам его нельзя непосредственно встраивать в контейнер с элементами, располагаемыми по сетке, – для использования его в таком контексте добавьте промежуточный `Frame` с сеткой.

Классы GuiMaker

В ответ на выполнение условий по протоколам и шаблонам `GuiMaker` клиентские подклассы получают `Frame`, который умеет автоматически строить свои меню и панели инструментов по структурам данных шаблона. Если вы смотрели примеры меню из предыдущей главы, то можете понять, что это большое приобретение в смысле уменьшения объема кода. `GuiMaker` также достаточно сообразителен и может экспортировать интерфейсы меню обоих стилей, с которыми мы встречались в предыдущей главе:

- `GuiMakerWindowMenu` реализует меню окон верхнего уровня в стиле Tk 8.0, применяемые для меню, связываемых с самостоятельными программами, и всплывающих окон.
- `GuiMakerFrameMenu` реализует альтернативные меню, основанные на `Frame/Menubutton`, применяемые для меню объектов, встраиваемых в качестве компонентов более крупных GUI.

Оба класса строят панели инструментов, экспортируют те же протоколы и ожидают те же структуры шаблонов; они отличаются только способом обработки шаблонов меню. В действительности один из них является подклассом другого, специализируя метод создания меню – два стиля отличаются только обработкой меню верхнего уровня (`Menu` с каскадами `Menu` вместо `Frame` с `Menubuttons`).

Самотестирование GuiMaker

Как и в GuiMixin, при выполнении примера 9.2 в качестве программы верхнего уровня запускается логика самотестирования, находящаяся в конце файла. На рис. 9.2 показаны получаемые при этом окна. Возникают три окна, представляющие классы TestApp кода самотестирования. У всех трех есть меню и панель инструментов, параметры которых заданы в структурах данных шаблонов, создаваемых кодом самотестирования: выпадающие меню File и Edit, а также кнопка панели инструментов Quit и стандартная кнопка меню Help. На снимке меню File одного из окон оторвано, а меню Edit другого окна раскрыто.

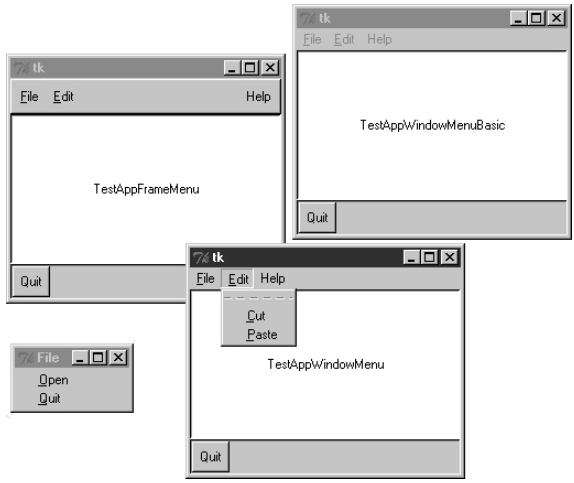


Рис. 9.2. Самотестирование GuiMaker в работе

Связи между суперклассами установлены в коде так, что два из трех окон получают обработчик обратного вызова help от GuiMixin; TestAppWindowMenuBasic получает его от GuiMaker. Обратите внимание на то, что важен порядок, в котором соединяются эти два класса: так как метод quit определен в обоих классах GuiMixin и Frame, класс, от которого мы хотим его получить, нужно указать первым в строке заголовка смешанного класса, поскольку при множественном наследовании поиск производится слева направо. Чтобы выбирались методы GuiMixin, его обычно следует указывать перед суперклассом, производным от реальных графических элементов.

Более практическое применение GuiMaker найдет в таких примерах, как PyEdit далее в этой главе. В следующем модуле показан другой способ использования шаблонов GuiMaker для построения усложненного интерфейса.

BigGui: программа демонстрации клиента

Рассмотрим программу, в которой лучшим образом используются только что написанные нами два класса автоматизации. В модуле примера 9.3 класс Hello является наследником как GuiMixin, так и GuiMaker. GuiMaker обеспечивает связь с графическим элементом Frame и логику построения меню/панели инструментов. GuiMixin обеспечивает дополнительные методы стандартного поведения. В действительности Hello служит образцом еще одного расширения элемента Frame, поскольку является производным от GuiMaker. Чтобы бесплатно получить меню и панель инструментов, он просто следует протоколам, определенным в GuiMaker, – устанавливает атрибуты menuBar и toolBar в методе start и переопределяет makeWidgets, помещая в середину метку.

Пример 9.3. PP2E\Gui\Tools\BigGui\big_gui.py

```
#!/usr/bin/python
###########################################################
# реализация gui - объединяет maker, mixin и данный
###########################################################

import sys, os, string
from Tkinter import *                          # классы графических элементов
from PP2E.Gui.Tools.guimixin import *          # внедряемые методы
from PP2E.Gui.Tools.guimaker import *          # фрейм плюс построение меню/панели инструментов
from find_demo_dir import findDemoDir          # поиск демопрограмм Python

class Hello(GuiMixin, GuiMakerWindowMenu):     # или GuiMakerFrameMenu
    def start(self):
        self.hellos = 0
        self.master.title("GuiMaker Demo")
        self.master.iconname("GuiMaker")

        self.menuBar = [                               # дерево: 3 выпадающих меню
          ('File', 0,                                  # (выпадающее меню)
              [('New...',  0, self.notdone),           # [список элементов меню]
               ('Open...', 0, self.fileOpen),
               ('Quit',    0, self.quit)]              # метка,подчеркивание,действие
          ),

          ('Edit', 0,
              [('Cut',  -1, self.notdone),             # без подчеркивания|действия
               ('Paste',-1, self.notdone),             # lambda:0 тоже можно
               'separator',                            # добавить разделитель
               ('Stuff', -1,
                   [('Clone', -1, self.clone),         # каскадное подменю
                    ('More',  -1, self.more)]
               ),
               ('Delete', -1, lambda:0),
               [5]]                                    # отключить 'delete'
          ),

          ('Play', 0,
              [('Hello',     0, self.greeting),
               ('Popup...',  0, self.dialog),
               ('Demos', 0,
                  [('Hanoi', 0,
                       lambda x=self:
                        x.spawn(findDemoDir() + '\guido\hanoi.py', wait=0)),
                   ('Pong',  0,
                       lambda x=self:
                       x.spawn(findDemoDir() + '\matt\pong-demo-1.py', wait=0)),
                   ('Other...', -1, self.pickDemo)]
               )]
          )]

        self.toolBar = [
          ('Quit',  self.quit,     {'side': RIGHT}),       # добавить 3 кнопки
          ('Hello', self.greeting, {'side': LEFT}),
          ('Popup', self.dialog,   {'side': LEFT, 'expand':YES}) ]

    def makeWidgets(self):                             # переопределить имеющийся по умолчанию
        middle = Label(self, text='Hello maker world!', width=40, height=10,
                       cursor='pencil', bg='white', relief=SUNKEN)
        middle.pack(expand=YES, fill=BOTH)
```

```
    def greeting(self):
        self.hellos = self.hellos + 1
        if self.hellos % 3:
            print "hi"
        else:
            self.infobox("Three", 'HELLO!')    # при каждом третьем нажатии

    def dialog(self):
        button = self.question('OOPS!',
                               'You typed "rm*" ... continue?',
                               'questhead', ('yes', 'no', 'help'))
        [lambda:0, self.quit, self.help][button]()

    def fileOpen(self):
        pick = self.selectOpenFile(file='big_gui.py')
        if pick:
            self.browser(pick)     # просмотр файла исходного кода или иного

    def more(self):
        new = Toplevel()
        Label(new,  text='A new non-modal window').pack()
        Button(new, text='Quit', command=self.quit).pack(side=LEFT)
        Button(new, text='More', command=self.more).pack(side=RIGHT)

    def pickDemo(self):
        pick = self.selectOpenFile(dir=findDemoDir()+'\guido')
        if pick:
            self.spawn(pick, wait=0)    # запустить любую программу Python

if __name__ == '__main__':  Hello().mainloop()   # создать, запустить
```

Этот сценарий размещает довольно большую структуру из меню и панели инструментов, которую мы вскоре увидим. Он также добавляет собственные методы обратного вызова, которые выводят сообщения в `stdout`, показывают средства просмотра текстовых файлов и новые окна и запускают программы. Однако многие обратные вызовы не делают ничего, кроме запуска метода `notDone`, унаследованного от `GuiMixin`: этот код предназначен в основном для демонстрации `GuiMaker` и `GuiMixin`.

Сценарий `big_gui` является почти законченной программой, но не совсем: он нуждается во вспомогательном модуле для поиска готовых демонстрационных программ, поставляемых вместе с полным дистрибутивом исходного кода Python. (Эти демонстрационные программы не входят в собрание примеров данной книги.) Дистрибутив исходного кода Python может быть распакован в любом месте машины.

По этой причине неизвестно, где располагается каталог с демонстрационными программами (и есть ли он вообще). Но вместо того чтобы предложить начинающим изменить исходный код сценария, задав в нем соответствующий путь, выполняется поиск демонстрационного каталога с помощью инструмента `guessLocation` из модуля `Launcher`, с которым мы познакомились в конце главы 4 (см. пример 9.4). Если вы забыли, как это действует, перелистайте книгу назад (хотя прелесть повторного использования кода заключается в том, что часто можно позволить себе забывать).

Пример 9.4. PP2E\Gui\Tools\BigGui\find_demo_dir.py

```
################################################################################
# поиск демонстрационных программ, распространяемых с дистрибутивом исходного кода Python;
# PATH и PP2EHOME здесь не помогут, т. к. эти программы не входят в стандартную установку
# или дерево данной книги
################################################################################
```

```
import os, string, PP2E.Launcher
demoDir  = None
myTryDir = ''

#sourceDir = r'C:\Stuff\Etc\Python-ddj-cd\distributions'
#myTryDir  = sourceDir + r'\Python-1.5.2\Demo\tkinter'

def findDemoDir():
    global demoDir
    if not demoDir:                          # поиск только при первом обращении
        if os.path.exists(myTryDir):         # использовать каталог, заданный в коде, или искать
            demoDir = myTryDir               # сохранить в глобальной переменной
                                             # для следующего обращения
        else:
            print 'Searching for standard demos on your machine...'
            path = PP2E.Launcher.guessLocation('hanoi.py')
            if path:
                demoDir = string.join(string.split(path, os.sep)[:-2], os.sep)
                print 'Using demo dir:', demoDir
    assert demoDir, 'Where is your demo directory?'
    return demoDir
```

При выполнении `big_gui` в качестве программы верхнего уровня создается окно с четырьмя выпадающими меню вверху и панелью инструментов с тремя кнопками внизу, которое показано на рис. 9.3 вместе с некоторыми всплывающими окнами, созданными его обратными вызовами. В меню есть разделители, отключенные элементы и каскадные подменю в полном соответствии с шаблоном `menuBar`.

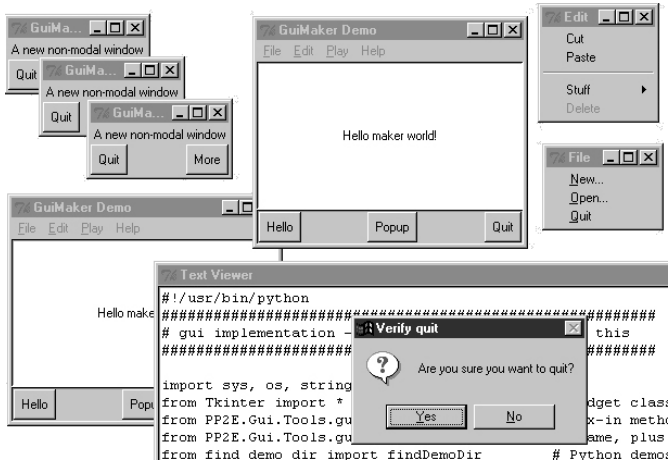


Рис. 9.3. big_gui с несколькими всплывающими окнами

Рис. 9.4 снова показывает окно этого сценария после того, как через выпадающее меню Play были запущены два независимо выполняющиеся экземпляра сценария *hanoi.py*, поставляемого с дистрибутивом исходного кода Python и написанного создателем Python Гвидо ван Россумом. Эта демонстрационная программа содержит простую анимацию решения головоломки «ханойская башня» – классической рекурсивной задачи, часто встречающейся в контрольных опросах по информатике (если вы никогда о ней не слышали, я пощажу вас от изложения деталей).

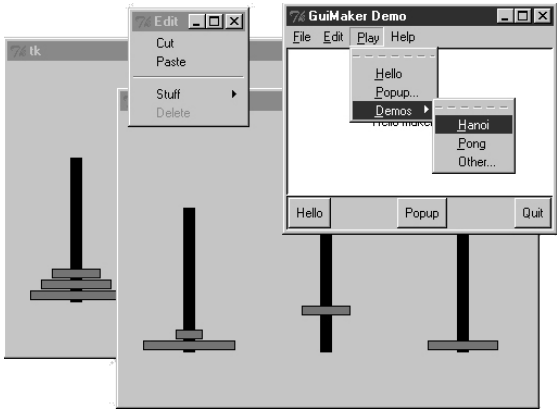


Рис. 9.4. big_gui с порожденными демонстрационными программами hanoi в процессе выполнения

Чтобы обнаружить эту демонстрационную программу, сценарий обыскивает деревья каталогов машины, расположенные в стандартных местах; на моей машине его удалось найти, только прибегнув в последнюю очередь к обходу всего жесткого диска *C:*:

```
C:\...\PP2E\Gui\Tools\BigGui>python big_gui.py
Searching for standard demos on your machine...
Searching for hanoi.py in C:\Program Files\Python
Searching for hanoi.py in C:\PP2ndEd\examples\PP2E\Gui\Tools\BigGui
Searching for hanoi.py in C:\Program Files
Searching for hanoi.py in C:\
Using demo dir: C:\PP2ndEd\cdrom\Python1.5.2\SourceDistribution\Unpacked\Python-
1.5.2\Demo\tkinter
C:\PP2ndEd\cdrom\Python1.5.2\SourceDistribution\Unpacked\Python-1.5.2\Demo\tkint
er\guido\hanoi.py
```

Этот поиск длится примерно 20 секунд на моем 650-мегагерцовом переносном компьютере под Windows, но выполняется только при первом выборе одной из этих демонстрационных программ – после успешного поиска модуль `find_demo_dir` сохраняет имя каталога в глобальной переменной, чтобы его можно было немедленно извлечь при следующем вызове демонстрационной программы. Если нужно выполнить демонстрационные программы из других каталогов (например, программ этой книги в дереве *PP2E*), выберите в меню Play вариант Other, который вызовет стандартный диалог выбора файла, и найдите файл нужной программы.

Наконец, хочу заметить, что `GuiMaker` можно перепроектировать так, чтобы в нем использовались деревья вложенных экземпляров классов, умеющих применять себя к строящемуся дереву графических элементов Tkinter, а не ветвление по типам элементов в структурах данных шаблона. Однако ввиду недостатка места мы отнесем это расширение к области предлагаемых в данном издании самостоятельных упражнений.

ShellGui: добавление GUI к инструментам командной строки

Чтобы явственнее продемонстрировать практическую пользу таких конструкций, как класс `GuiMixin`, требуется более реальное приложение. Вот одно из них. В главе 4 мы познакомились с простыми сценариями для упаковки и распаковки текстовых файлов (см. раздел «Упаковка и распаковка файлов»). Там, если помните, сценарий

packapp.py конкатенировал несколько текстовых файлов в один, а *unpackapp.py* извлекал исходные файлы из объединенного файла.

Эти сценарии запускались с помощью вводимых вручную командных строк, не самых сложных, которые могут встретиться, но достаточно непростых, чтобы легко забыть их. Вместо того чтобы требовать от пользователей таких утилит вводить загадочные команды в оболочке, почему бы для запуска этих программ не создать простой в использовании интерфейс Tkinter? И если мы ставим такую задачу, почему бы вообще не обобщить идею запуска утилит командной строки из GUI для обеспечения поддержки и тех утилит, которые появятся в будущем?

Общий экран инструментов оболочки

Примеры с 9.5 по 9.8 представляют одну конкретную реализацию этих абстрактных размышлений. Так как я хотел, чтобы это был инструмент общего назначения, способный запустить любую программу командной строки, его конструкция разбита на модули, которые становятся все более специфическими для приложений по мере более глубокого вхождения в иерархию программного обеспечения. На самом верху все является общим настолько, насколько это возможно, как показано в примере 9.5.

Пример 9.5. PP2E\Gui\ShellGui\shellgui.py.py

```
#!/usr/local/bin/python
################################################################################
# запуск утилит; использует шаблоны guimaker, стандартный диалог завершения guimixin;
# это лишь библиотека классов: вывести gui можно сценарием mytools;
################################################################################

from Tkinter import *                               # получить графические элементы
from PP2E.Gui.Tools.guimixin import GuiMixin        # получить quit, not done
from PP2E.Gui.Tools.guimaker import *               # построитель меню/панелей инструментов

class ShellGui(GuiMixin, GuiMakerWindowMenu):       # фрейм + конструктор + mixin
    def start(self):                                # для компонентов использовать GuiMaker
        self.setMenuBar()
        self.setToolBar()
        self.master.title("Shell Tools Listbox")
        self.master.iconname("Shell Tools")

    def handleList(self, event):                    # при двойном щелчке по окну списка
        label = self.listbox.get(ACTIVE)            # получить выбранный текст
        self.runCommand(label)                      # и вызвать действие

    def makeWidgets(self):                          # поместить в середину окно списка
        sbar = Scrollbar(self)                      # связать sbar со списком
        list = Listbox(self, bg='white')            # или использовать Tour.ScrolledList
        sbar.config(command=list.yview)
        list.config(yscrollcommand=sbar.set)
        sbar.pack(side=RIGHT, fill=Y)                   # упаковать первым - обрезать последним
        list.pack(side=LEFT, expand=YES, fill=BOTH)     # первым обрезается список
        for (label, action) in self.fetchCommands():    # добавить в окно списка
            list.insert(END, label)                     # и меню/панель инструментов
        list.bind('<Double-1>', self.handleList)        # установить обработчик события
        self.listbox = list

    def forToolBar(self, label):                    # поместить на панель инструментов?
        return 1                                    # по умолчанию - все

    def setToolBar(self):
        self.toolBar = []
```

```
        for (label, action) in self.fetchCommands():
            if self.forToolBar(label):
                self.toolBar.append((label, action, {'side': LEFT}))
        self.toolBar.append(('Quit', self.quit, {'side': RIGHT}))

    def setMenuBar(self):
        toolEntries  = []
        self.menuBar = [
            ('File',  0, [('Quit', -1, self.quit)]),     # имя выпадающего меню
            ('Tools', 0, toolEntries)                    # список элементов меню
            ]                                            # метка, подчеркивание
                                                         # для быстрого вызова, действие
        for (label, action) in self.fetchCommands():
            toolEntries.append((label, -1, action))      # добавить приложения элементами меню

##########################################################################
# делегирование в шаблонные, специфичные для типов, подклассы,
# которые делегируют в подклассы, специфичные для набора утилит
##########################################################################

class ListMenuGui(ShellGui):
    def fetchCommands(self):              # подкласс: установка 'MyMenu'
        return self.myMenu                # список вида (метка, функция)
    def runCommand(self, cmd):
        for (label, action) in self.myMenu:
            if label == cmd: action()

class DictMenuGui(ShellGui):
    def fetchCommands(self):   return self.myMenu.items()
    def runCommand(self, cmd): self.myMenu[cmd]()
```

Класс `ShellGui`, находящийся в этом модуле, умеет с помощью интерфейсов `GuiMaker` и `GuiMixin` строить окно для выбора, которое выводит имена утилит в меню, прокручиваемом списке и панели инструментов. Он также предоставляет переопределяемый метод `forToolBar`, позволяющий подклассам указывать, какие утилиты должны добавляться в панель инструментов, а какие – нет (на панели инструментов может быстро не оказаться свободного места). Однако он умышленно оставлен в неведении относительно имен утилит, которые должны быть выведены в указанных местах, и действий, которые должны быть выполнены при выборе имен утилит.

Вместо этого `ShellGui` использует находящиеся в этом файле подклассы `ListMenuGui` и `DictMenuGui`, чтобы получить список имен утилит через метод `fetchCommands` и управлять действиями по именам через метод `runCommand`. Эти два подкласса в действительности лишь предоставляют интерфейс к специфическим для приложения наборам утилит, представленным как списки или словари; они по-прежнему не знают, какие имена утилит в действительности появляются в GUI. Это также сделано умышленно: так как отображаемые наборы утилит определяются подклассами более низкого уровня, с помощью `ShellGui` можно отображать различные наборы утилит.

Классы наборов утилит, специфичные для приложений

Чтобы получать фактические наборы утилит, нужно спуститься вниз на один уровень. Модуль примера 9.6 определяет подклассы двух специфических по типу классов `ShellGui`, чтобы предоставить имеющиеся утилиты в формате как списка, так и словаря (обычно требуется только один из них, но модуль служит для иллюстрации обоих). Это также тот модуль, который действительно *выполняется* для запуска GUI – модуль `shellgui` служит только библиотекой классов.

Пример 9.6. PP2E\Gui\ShellGui\mytools.py

```
#!/usr/local/bin/python
from shellgui import *                  # специфичные по типу интерфейсы оболочки
from packdlg  import runPackDialog      # диалоги для ввода данных
from unpkdlg  import runUnpackDialog    # оба выполняют классы приложений

class TextPak1(ListMenuGui):
    def __init__(self):
        self.myMenu = [('Pack',    runPackDialog),
                       ('Unpack',  runUnpackDialog),    # простые функции
                       ('Mtool',   self.notdone)]       # метод из guimixin
        ListMenuGui.__init__(self)

    def forToolBar(self, label):
        return label in ['Pack', 'Unpack']

class TextPak2(DictMenuGui):
    def __init__(self):
        self.myMenu = {'Pack':    runPackDialog,        # или использовать входные
                       'Unpack':  runUnpackDialog,    # данные вместо диалогов
                       'Mtool':   self.notdone}
        DictMenuGui.__init__(self)

if __name__ == '__main__':                             # код самопроверки...
    from sys import argv                               # 'menugui.py list|^'
    if len(argv) > 1 and argv[1] == 'list':
        print 'list test'
        TextPak1().mainloop()
    else:
        print 'dict test'
        TextPak2().mainloop()
```

Классы в этом модуле специфичны для конкретных наборов утилит; чтобы выводить другой набор имен утилит, нужно написать новый подкласс и выполнить его. Разделение логики приложения на такие отдельные подклассы и модули повышает возможность повторного использования программного обеспечения.

На рис. 9.5 показано главное окно ShellGui, создаваемое при запуске под Windows сценария mytools с классом структуры меню, основывающемся на словаре, а также оторванные меню, демонстрирующие свое содержание. Меню этого окна и его панель

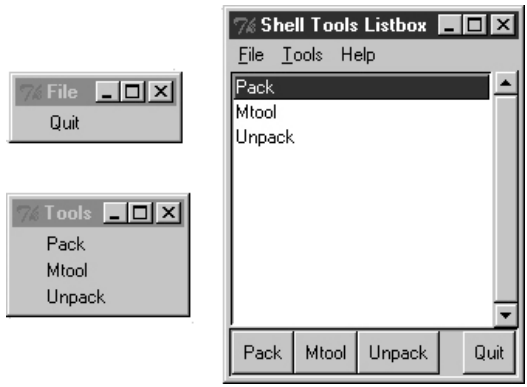


Рис. 9.5. Элементы mytools в окне ShellGui

инструментов построил GuiMaker, а кнопки Quit и Help и выбор пунктов меню запускают методы quit и help, унаследованные от GuiMixin через суперклассы модуля ShellGui. Вы начинаете понимать, почему в этой книге столь часто проповедуется повторное использование кода?

Добавление GUI клиента к командным строкам

Однако действия в обратных вызовах, задаваемые классами предыдущего модуля, обычно должны выполнять нечто GUI-ориентированное. Так как миром исходных сценариев упаковки и распаковки файлов являются текстовые потоки, их нужно заключить в кодовые оболочки, принимающие входные параметры от пользователей, настроенных на работу с GUI.

Модуль примера 9.7 использует технику пользовательских модальных диалогов, изученную нами в главе 7 «Обзор Tkinter, часть 1», чтобы вывести экран для получения параметров сценария упаковки. Его функция runPackDialog служит фактическим обработчиком обратного вызова, запускаемым при выборе имен утилит в главном окне ShellGui.

Пример 9.7. PP2E\Gui\ShellGui\packdlg.py.

```
# добавлены диалоги выбора файлов, проверка пустых полей; можно использовать сетку

import string
from glob import glob                                   # расширение имен файлов
from Tkinter import *                                   # графические элементы для gui
from tkFileDialog import *                              # диалог выбора файла
from PP2E.System.App.Clients.packapp import PackApp     # упаковывающий класс

def runPackDialog():
    s1, s2 = StringVar(), StringVar()          # запуск класса как функции
    PackDialog(s1, s2)                         # всплывающий диалог: устанавливает s1/s2
    output, patterns = s1.get(), s2.get()      # 'ok' или wm-destroy
    if output != "" and patterns != "":
        patterns = string.split(patterns)
        filenames = []
        for sublist in map(glob, patterns):    # расширение вручную
            filenames = filenames + sublist    # автоматически в командной строке Unix
        print 'PackApp:', output, filenames
        app = PackApp(ofile=output)            # запуск с переадресацией вывода
        app.args = filenames                   # переустановка списка аргументов
        app.main()                             # в gui тоже должны показываться сообщения

class PackDialog(Toplevel):
    def __init__(self, target1, target2):
        Toplevel.__init__(self)                # новое окно верхнего уровня
        self.title('Enter Pack Parameters')    # 2 фрейма и кнопка

        f1 = Frame(self)
        l1 = Label(f1,  text='Output file?', relief=RIDGE, width=15)
        e1 = Entry(f1,  relief=SUNKEN)
        b1 = Button(f1, text='browse...')
        f1.pack(fill=X)
        l1.pack(side=LEFT)
        e1.pack(side=LEFT, expand=YES, fill=X)
        b1.pack(side=RIGHT)
        b1.config(command= (lambda x=target1: x.set(askopenfilename())) )

        f2 = Frame(self)
        l2 = Label(f2,  text='Files to pack?', relief=RIDGE, width=15)
```

```
        e2 = Entry(f2,  relief=SUNKEN)
        b2 = Button(f2, text='browse...')
        f2.pack(fill=X)
        l2.pack(side=LEFT)
        e2.pack(side=LEFT, expand=YES, fill=X)
        b2.pack(side=RIGHT)
        b2.config(command=
                 (lambda x=target2: x.set(x.get() +' '+ askopenfilename())) )

        Button(self, text='OK', command=self.destroy).pack()
        e1.config(textvariable=target1)
        e2.config(textvariable=target2)

        self.grab_set()          # сделать модальным: захват мыши,
        self.focus_set()         # фокус клавиатуры, ожидание...
        self.wait_window()       # до закрытия; иначе сразу возврат

if __name__ == '__main__':
    root = Tk()
    Button(root, text='pop', command=runPackDialog).pack(fill=X)
    Button(root, text='bye', command=root.quit).pack(fill=X)
    root.mainloop()
```

При выполнении этого сценария создается форма для ввода, показанная на рис. 9.6. Пользователь может ввести имена входных и выходных файлов с клавиатуры или нажать кнопку «browse...» для открытия стандартных диалогов выбора файлов. Можно вводить и шаблоны имен файлов – вызов `glob.glob` вручную расширяет шаблоны имен файлов и отфильтровывает имена несуществующих файлов. Командная строка Unix осуществляет такое расширение шаблонов автоматически при запуске `PackApp` из оболочки, в отличие от Windows (см. подробности в главе 2 «Системные инструменты»).

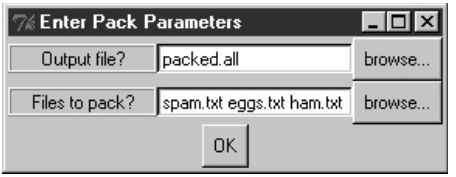


Рис. 9.6. Форма ввода packdlg

Когда форма заполнена и послана с помощью кнопки OK, параметры передаются экземпляру созданного нами в главе 4 класса `PackApp`, чтобы выполнить конкатенацию файлов. GUI для сценария распаковки проще, так как в нем только одно поле ввода – для имени упакованного файла. Сценарий примера 9.8 создает окно формы ввода, показанное на рис. 9.7.

Пример 9.8. PP2E\Gui\ShellGui\unpkdlg.py

```
# добавлен диалог выбора файла, лучше обрабатывает отмену

from Tkinter import *                                   # классы графических элементов
from tkFileDialog import *                              # диалог открытия файла
from PP2E.System.App.Clients.unpackapp import UnpackApp     # класс распаковщика

def runUnpackDialog():
    input = UnpackDialog().input                        # получить входные данные из GUI
    if input != '':                                     # работа с файлами вне gui
        print 'UnpackApp:', input
        app = UnpackApp(ifile=input)                    # запустить с входным файлом
```

```
        app.main()                                          # выполнить класс app

class UnpackDialog(Toplevel):
    def __init__(self):                                     # функция тоже работает
        Toplevel.__init__(self)                             # корневое окно
        self.input = ''                                     # метка и поле ввода
        self.title('Enter Unpack Parameters')
        Label(self, text='input file?', relief=RIDGE, width=11).pack(side=LEFT)
        e = Entry(self, relief=SUNKEN)
        b = Button(self, text='browse...')
        e.bind('<Key-Return>', self.gotit)
        b.config(command=(lambda x=e: x.insert(0, askopenfilename())))
        b.pack(side=RIGHT)
        e.pack(side=LEFT, expand=YES, fill=X)
        self.entry = e
        self.grab_set()                                 # сделать модальным
        self.focus_set()
        self.wait_window()                              # до закрытия по return->gotit
    def gotit(self, event):                             # по клавише return: event.widget==Entry
        self.input = self.entry.get()                   # получить текст, сохранить в self
        self.destroy()                                  # убить окно, но экземпляр сохраняется

if __name__ == "__main__":
    Button(None, text='pop', command=runUnpackDialog).pack()
    mainloop()
```

Кнопка «browse...» на рис. 9.7 показывает диалог выбора файла так же, как форма `packdlg`. Вместо кнопки OK этот диалог привязывает событие нажатия клавиши Enter к закрытию окна и завершению паузы, вызванной ожиданием в модальном состоянии; при этом имя файла передается экземпляру класса `UnpackApp` из главы 4 для выполнения реальной процедуры сканирования файла.

Рис. 9.7. Форма ввода unpkdlg

Все это действует так, как обещано, – благодаря такой доступности утилит командной строки в графическом виде они становятся значительно более привлекательными для пользователей, привычным образом жизни которых является GUI. Все же есть два аспекта такой конструкции, которые представляются требующими усовершенствования.

Во-первых, оба диалога для ввода содержат специальный код для передачи особого внешнего вида, однако кажется возможным существенно упростить их путем импорта стандартного модуля для построения форм. Мы встречались с обобщенным кодом для построения форм в главах 7 и 8 и будем встречаться с ним и далее; см. также указания по обобщению создания форм в модуле *form.py* в главе 10 «Сетевые сценарии».

Во-вторых, в тот момент, когда пользователь передает данные, введенные в той или иной диалоговой форме, след GUI теряется – сообщения `PackApp` и `UnpackApp` по-прежнему поступают в окно консоли `stdout`:

```
C:\...\PP2E\Gui\ShellGui\test>python ..\mytools.py
dict test
PackApp: packed.all ['spam.txt', 'eggs.txt', 'ham.txt']
packing: spam.txt
packing: eggs.txt
packing: ham.txt
```

```
UnpackApp: packed.all
creating: spam.txt
creating: eggs.txt
creating: ham.txt
```

Лучшим решением будет *переадресация* stdout в объект, который выводит полученный текст в окно GUI. О том, как это сделать, можно прочесть в следующем разделе.

GuiStreams: перенаправление потоков данных элементам GUI

Сценарий в примере 9.9 организует отображение входных и выходных потоков во всплывающие окна приложения GUI способом, во многом напоминающим то, как мы поступали со строками в темах, связанных с переадресацией потоков в главе 2. Несмотря на то что этот модуль служит лишь начальным прототипом и сам нуждается в усовершенствовании (например, каждый запрос строки ввода появляется в новом диалоговом окне ввода), он демонстрирует идею в целом.

Объекты этого модуля GuiOutput и GuiInput определяют методы, позволяющие им маскироваться под файлы в любом интерфейсе, предполагающем файл. Как известно из главы 2, это осуществляется с помощью средств обработки стандартных потоков, таких как print, raw_input, и явных вызовов read и write. Функция redirectedGuiFunc из этого модуля с помощью такой «plug-and-play»-совместимости позволяет выполнять любую функцию так, что ее стандартные входной и выходной потоки целиком отображаются во всплывающие окна, а не в окно консоли (или туда, куда эти потоки отображались бы в обычном случае).

Пример 9.9. PP2E\Gui\Tools\guiStreams.py

```
##############################################################################################
# начальная реализация классов, похожих на файлы, используемых для переадресации
# входных и выходных потоков в экраны GUI; входные данные поступают из стандартного
# диалога (единый интерфейс output+input или постоянное поле Entry
# для ввода было бы лучше); кроме того, неверно берутся строки в запросах чтения
# при количестве байтов > len(строки); см. также guiStreamsTools.py;
##############################################################################################
from Tkinter import *
from ScrolledText import ScrolledText
from tkSimpleDialog import askstring

class GuiOutput:
    def __init__(self, parent=None):
        self.text = None
        if parent: self.popupnow(parent)          # вывод сейчас или при первой записи
    def popupnow(self, parent=None):              # сейчас в родительском, Toplevel потом
        if self.text: return
        self.text = ScrolledText(parent or Toplevel())
        self.text.config(font=('courier', 9, 'normal'))
        self.text.pack()
    def write(self, text):
        self.popupnow()
        self.text.insert(END, str(text))
        self.text.see(END)
        self.text.update()
    def writelines(self, lines):                  # в строках уже есть '\n'
        for line in lines: self.write(line)       # или map(self.write, lines)

class GuiInput:
    def __init__(self):
```

```
        self.buff = ''
    def inputLine(self):
        line = askstring('GuiInput', 'Enter input line + <crlf> (cancel=eof)')
        if line == None:
            return ''                       # всплывающий диалог для каждой строки
        else:                               # кнопка cancel означает eof
            return line + '\n'              # иначе добавить маркер конца строки
    def read(self, bytes=None):
        if not self.buff:
            self.buff = self.inputLine()
        if bytes:                                   # читать по счетчику байтов
            text = self.buff[:bytes]                # не захватывает строки
            self.buff = self.buff[bytes:]
        else:
            text = ''                               # читать все до eof
            line = self.buff
            while line:
                text = text + line
                line = self.inputLine()             # до cancel=eof=''
        return text
    def readline(self):
        text = self.buff or self.inputLine()        # эмуляция методов чтения файлов
        self.buff = ''
        return text
    def readlines(self):
        lines = []                                  # читать все строки
        while 1:
            next = self.readline()
            if not next: break
            lines.append(next)
        return lines

def redirectedGuiFunc(func, *pargs, **kargs):
    import sys
    saveStreams = sys.stdin, sys.stdout         # отображение потоков func во всплывающие окна
    sys.stdin   = GuiInput()                    # показывает нужный диалог
    sys.stdout  = GuiOutput()                   # новое окно вывода для каждого вызова
    sys.stderr  = sys.stdout
    result = apply(func, pargs, kargs)          # это блокирующий вызов func
    sys.stdin, sys.stdout = saveStreams
    return result

def redirectedGUIhellCmd(command):
    import os
    input  = os.popen(command, 'r')
    output = GuiOutput()
    def reader(input, output):                  # показать стандартный вывод
        while 1:                                # команды оболочки в новом
            line = input.readline()             # элементе текстового окна
            if not line: break
            output.write(line)
    reader(input, output)

if __name__ == '__main__':
    import string
    def makeUpper():                            # использовать стандартные потоки
        while 1:
            try:
                line = raw_input('Line? ')
```

```
            except:
                break
            print string.upper(line)
        print 'end of file'

    def makeLower(input, output):                # использовать явные файлы
        while 1:
            line = input.readline()
            if not line: break
            output.write(string.lower(line))
        print 'end of file'

    root = Tk()
    Button(root, text='test streams',
           command=lambda: redirectedGuiFunc(makeUpper)).pack(fill=X)
    Button(root, text='test files  ',
           command=lambda: makeLower(GuiInput(), GuiOutput()) ).pack(fill=X)
    Button(root, text='test popen  ',
           command=lambda: redirectedGUIhellCmd('dir *')).pack(fill=X)
    root.mainloop()
```

Согласно этому коду `GuiOutput` либо прикрепляет `ScrolledText` к родительскому контейнеру, либо выводит новое окно верхнего уровня, которое должно служить контейнером, при первом обращении к записи. `GuiInput` выводит новый стандартный диалог ввода каждый раз, когда запрос чтения требует новую входную строку. Ни одна из этих схем не является идеальной во всех ситуациях (ввод лучше было бы отобразить на более долговременный графический элемент), но они доказывают правильность общей идеи. На рис. 9.8 показана картина, создаваемая кодом самотестирования этого сценария, после перехвата вывода команды оболочки *dir* (слева) и двух интерактивных проверок цикла (окно с приглашениями «Line?» и заглавными буквами представляет тест потоков `makeUpper`). Диалог для ввода выведен для нового теста файлов `makeLower`.

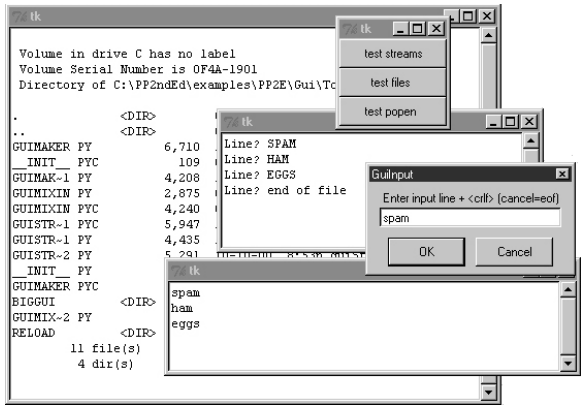


Рис. 9.8. guiStreams направляет потоки во всплывающие окна

Использование переадресации со сценариями упаковки

Теперь, чтобы использовать эти инструменты переадресации для отображения вывода сценария командной строки обратно в GUI, просто выполним вызовы и командные строки через две переадресованные функции этого модуля. Пример 9.10 показывает один из способов создания оболочки для операции упаковки, благодаря которой вывод операции оказывается во всплывающем окне вместо консоли.

Пример 9.10. PP2E\Gui\ShellGui\packdlg-redirect.py

```
# оболочка сценария командной строки в виде утилиты GUI,
# переадресующей его вывод во всплывающее окно

from Tkinter import *
from packdlg import runPackDialog
from PP2E.Gui.Tools.GUIstreams import redirectedGuiFunc

def runPackDialog_Wrapped():
    redirectedGuiFunc(runPackDialog)    # оболочка для всего обработчика обратного вызова

if __name__ == '__main__':
    root = Tk()
    Button(root, text='pop', command=runPackDialog_Wrapped).pack(fill=X)
    root.mainloop()
```

Можете проверить работу этого сценария, вызвав его непосредственно, без привлечения окна `ShellGui`. На рис. 9.9 показано получающееся окно `stdout` после закрытия входного диалога упаковки. Окно появляется, как только сценарий создаст вывод, и предоставляет пользователю несколько более дружественный GUI, чем при отлове сообщений в консоли. Аналогичный код можно написать, чтобы диалог параметров распаковки тоже направлял свой вывод во всплывающее окно.[1] На самом деле с помощью такой техники можно направить во всплывающее окно вывод любого вызова функции или командной строки; как обычно, идея совместимых интерфейсов объектов в значительной мере обусловливает гибкость Python.

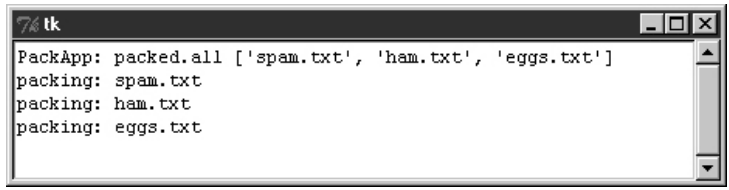


Рис. 9.9. Направление вывода сценария во всплывающее окно GUI

Динамическая перегрузка обработчиков обратного вызова GUI

Стоит рассмотреть здесь еще один последний прием программирования GUI. Функция Python `reload` позволяет динамически изменять и перегружать модули программы, не останавливая ее. Например, можно вызвать текстовый редактор, изменить отдельные части системы во время ее выполнения и увидеть, как проявляются эти изменения сразу после перегрузки измененного модуля.

Это мощная функция, особенно при разработке программ, перезапуск которых мог бы занять длительное время. Программы, которые подключаются к базам данных или сетевым серверам, инициализируют большие объекты или проходят долгую последовательность шагов, чтобы снова запустить обратный вызов, являются первыми кандидатами на использование `reload`. Эта функция может существенно сократить время разработки.

---

[1] Эти два примера довольно уникальны; так как применяемый в них суперкласс `App` сохраняет стандартные потоки в собственных атрибутах во время создания объекта, нужно запускать вызовы переадресующей оболочки GUI как можно раньше, чтобы `App` нашел переадресованные потоки GUI в `sys`, когда будет сохранять их локально. Большинство других сценариев не столь ловки в том, что касается внутренней переадресации потоков.

Однако особенностью GUI является то, что по причине регистрации обработчиков обратного вызова как *ссылок на объекты*, а не имен модулей и объектов, перегрузка функций обработчиков обратного вызова не действует после регистрации обратных вызовов. Операция Python `reload` действует путем изменения содержимого объекта модуля по месту. Однако из-за того что Tkinter непосредственно запоминает указатель на зарегистрированный объект обработчика, ему неведомо о перегрузке модуля, из которого происходит обработчик. Это означает, что Tkinter по-прежнему будет ссылаться на старые объекты модуля, даже если модуль изменен и перегружен.

Это тонкий момент, но нужно только запомнить, что для динамической перегрузки функций обработчиков обратного вызова требуется выполнить особые действия. Необходимо не только явно потребовать перегрузки измененных модулей, но и обеспечить некоторый косвенный слой, маршрутизирующий обратные вызовы от зарегистрированных объектов в модули, чтобы перегрузка возымела эффект.

Например, сценарий примера 9.11 делает дополнительную работу, косвенно переправляя обратные вызовы функциям в явно перегруженном модуле. Обработчики обратного вызова, зарегистрированные Tkinter, являются объектами методов, которые всего лишь осуществляют перегрузку и снова отправляют вызов. Так как доступ к действительным функциям обработчиков обратных вызовов происходит через объект модуля, перегрузка этого модуля приводит к обращению к последним версиям этих функций.

Пример 9.11. PP2E\Gui\Tools\Reload\rad.py

```
from Tkinter import *
import actions              # получить начальные обработчики обратных вызовов

class Hello(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.make_widgets()

    def make_widgets(self):
        Button(self, text='message1', command=self.message1).pack(side=LEFT)
        Button(self, text='message2', command=self.message2).pack(side=RIGHT)

    def message1(self):
        reload(actions)         # необходимо перегрузить перед вызовом модуль actions
        actions.message1()      # теперь нажатие кнопки запускает новую версию

    def message2(self):
        reload(actions)         # изменения actions.py действуют благодаря перегрузке
        actions.message2(self)  # вызвать последнюю версию; передать self

    def method1(self):
        print 'exposed method...'       # вызывается из функции actions

Hello().mainloop()
```

При выполнении этого сценария создается окно с двумя кнопками, запускающими `message1` и `message2`. Пример 9.12 содержит фактический код обработчика обратного вызова. Его функции получают аргумент `self`, который снова возвращает доступ к объекту класса `Hello`, как если бы это были реальные методы. Можно многократно изменять этот файл во время выполнения сценария `rad`; каждое такое действие изменяет поведение GUI при нажатии кнопки.

Пример 9.12. PP2E\Gui\Tools\Reload\actions.py

```
# обработчики обратного вызова: перегружаются при каждом запуске
```

```
def message1():                  # изменить себя
    print 'spamSpamSPAM'         # можно было построить диалог...

def message2(self):
    print 'Ni! Ni!'              # изменить себя
    self.method1()               # доступ к экземпляру 'Hello'...
```

Попробуйте запустить `rad` и редактировать сообщения, которые выводит `actions`, в другом окне. Вы должны увидеть, как при нажатии кнопок GUI в окно консоли `stdout` выводятся новые сообщения. Данный пример намеренно сделан простым, чтобы проиллюстрировать идею, но на практике перегружаемые таким способом действия могут создавать всплывающие диалоги, новые окна верхнего уровня и т. д. Перегрузка кода, создающего такие окна, позволяет динамически изменять их внешний вид.

Есть и другие способы изменить GUI во время его выполнения. Например, в главе 8 мы видели, что внешний вид в любой момент можно изменить, вызвав метод графического элемента `config`, а графические элементы можно динамически добавлять и удалять с экрана такими методами, как `pack_forget` и `pack` (и родственными им для менеджера окон `grid`). Кроме того, передача нового значения параметра `command=action` в метод `config` может на лету установить в качестве обработчика обратного вызова новый объект действия; при наличии соответствующего кода поддержки это может оказаться реальной альтернативой использованной выше обходной схеме повышения эффективности перегрузки в GUI.

Примеры законченных программ

В оставшейся части главы представлен ряд законченных программ GUI как примеров того, чего можно достичь с помощью Python и Tkinter. Так как я уже показывал интерфейсы, применяемые в этих сценариях, этот раздел состоит в основном из снимков экранов, листингов программ и нескольких абзацев, описывающих некоторые наиболее важные аспекты этих программ. Иными словами, это раздел для самостоятельного изучения: читайте исходный код, выполняйте примеры на своем компьютере и обращайтесь к предшествующим главам по вопросам, связанным с приведенным кодом. Многим из этих сценариев сопутствуют не приведенные здесь альтернативные или экспериментальные реализации на прилагаемом к книге CD; см. дополнительные программные примеры на CD.

PyEdit: программа/объект текстового редактора

За последние десятилетия мне пришлось набирать текст во многих программах. Большинство из них были закрытыми системами (мне приходилось довольствоваться теми решениями, которые были сделаны их разработчиками), и многие работали только на одной платформе. Представленная в этом разделе программа PyEdit более удачна в обоих отношениях: она реализует полноценный графический редактор текста примерно в 470 строках переносимого кода Python (включая пробельные символы и комментарии). Несмотря на свой размер, PyEdit оказался достаточно мощным и надежным, чтобы послужить основным инструментом для создания кода большинства примеров, приведенных в этой книге.

PyEdit поддерживает все обычные операции редактирования текста с помощью мыши и клавиатуры: удаление и вставка, поиск и замена, открытие и сохранение и т. д. Но в действительности PyEdit представляет собой нечто большее, чем просто текстовый редактор, – он разработан так, чтобы использоваться как программа и библиотечный компонент, и может выполняться в различном качестве:

Автономный режим

В качестве *автономной* программы текстового редактора, с возможностью передачи имени редактируемого файла в командной строке. В этом режиме PyEdit сходен с другими утилитами редактирования текста (например, Notepad в Windows), но, кроме того, предоставляет развитые функции, например запуск программы Python, код которой редактируется, изменение шрифта и цвета и т. д. Более важно то, что благодаря написанию PyEdit на Python его легко настраивать и переносимым образом выполнять в Windows, X Windows и Macintosh.

Всплывающий режим

Внутри нового всплывающего окна, позволяя программе одновременно выводить произвольное количество экземпляров. Поскольку информация о состоянии хранится в атрибутах экземпляра класса, каждый созданный объект PyEdit действует независимо. В этом режиме и в следующем PyEdit служит библиотечным объектом, используемым другими сценариями, а не готовым приложением.

Встроенный режим

В качестве *прикрепляемого* компонента, предоставляющего графический элемент редактирования текста для других GUI. Будучи прикреплен, PyEdit использует меню, основанное на фрейме, и может отключать некоторые опции меню. Например, PyView (далее в этой главе) использует PyEdit во встроенном режиме в качестве редактора подписей для фотографий, а PyMail (в главе 11) прикрепляет его и получает бесплатный редактор текста электронных писем.

Может показаться, что такое поведение с разными режимами трудно реализовать, но на самом деле режимы PyEdit по большей части являются побочными продуктами написания GUI с применением основанной на классах техники, рассматривавшейся на протяжении последних трех глав.

Запуск PyEdit

У PyEdit есть масса функций, и лучший способ понять, как он работает, – поработать с ним самостоятельно. Его можно запустить как файл *textEditor.pyw* или через панели запуска демонстрационных программ PyDemo и PyGadget, которые описаны в предыдущей главе (сами запускающие программы находятся на верхнем уровне дерева каталогов примеров книги). Получить представление о его интерфейсах можно из рис. 9.10, показывающему вид главного окна по умолчанию после открытия файла с исходным кодом PyEdit.

Главную часть этого окна составляет графический элемент `Text`, и если вы прочли его описание в предыдущей главе, то вам должны быть знакомы операции редактирования текста, выполняемые PyEdit. В нем используются метки, теги и индексы текста и реализованы операции удаления и вставки через системный буфер, благодаря чему PyEdit может обмениваться данными с другими приложениями. Для поддержки перемещения по произвольным файлам с элементом `Text` взаимно связаны вертикальная и горизонтальная полосы прокрутки.

Меню и панели инструментов PyEdit должны показаться вам знакомыми – он строит главное окно, используя минимальный объем кода и надлежащие схемы действий при обрезании и расширении, путем внедрения класса `GuiMaker`, с которым мы познакомились ранее в этой главе. Панель инструментов внизу окна содержит кнопки для сокращенного вызова операций, которым я пользуюсь чаще всего; если ваши вкусы не совпадают с моими, просто измените в исходном коде список для панели инструментов, чтобы в ней были те кнопки, которые вам нужны (в конце концов, это Python). Как обычно, в меню Tkinter для быстрого вызова элементов меню можно ис-

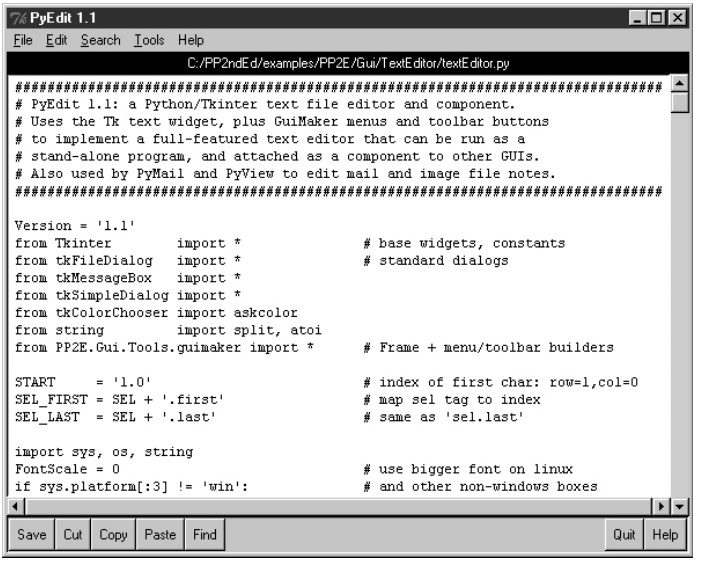


Рис. 9.10. Главное окно PyEdit, редактирующего самого себя

пользовать клавиши сокращенного набора – следует нажать <Alt> и все подчеркнутые клавиши на пути к нужному действию.

PyEdit выводит ряд модальных и немодальных диалогов, стандартных и пользовательских. Рис. 9.11 показывает пользовательский диалог замены и стандартный диалог для вывода статистики файла.

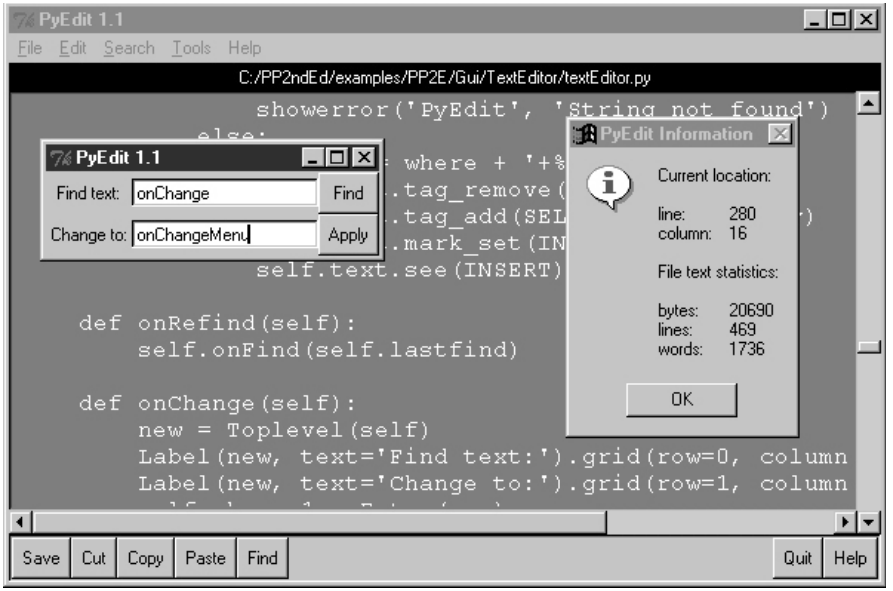


Рис. 9.11. PyEdit с цветом, шрифтами и некоторыми всплывающими диалогами

Здесь для главного окна установлены новые цвета переднего плана и фона (с помощью стандартного диалога выбора цвета), а для текста выбран новый шрифт из имеющегося в сценарии готового списка, который пользователи могут изменять в соответствии со своими предпочтениями (в конце концов, это Python). В PyEdit стандартные диалоги открытия и сохранения файла используют интерфейсы на основе объектов, которые запоминают каталог, выбиравшийся последним, и устраняют необходимость каждый раз заново находить к нему путь.

Одной из специфических особенностей PyEdit является возможность запуска программы Python, код которой редактируется. Это тоже не так сложно, как может показаться: поскольку в Python есть встроенные функции для компиляции и выполнения строк кода, а также запуска программ, PyEdit должен лишь выполнить соответствующие вызовы. В частности, легко написать на Python простенький интерпретатор Python (хотя осуществить обработку многострочных команд несколько сложнее), как показано в примере 9.13.

Пример 9.13. PP2E\Gui\TextEditor\simpleshell.py

```
namespace= {}
while 1:
    try:
        line = raw_input('>>> ')      # только однострочные предложения
    except EOFError:
        break
    else:
        exec line in namespace        # или eval() и print результата
```

В зависимости от настроек пользователя PyEdit либо делает что-нибудь аналогичное для выполнения кода, выбираемого из текстового элемента, либо использует модуль `launchmodes`, который мы написали в конце главы 3, чтобы выполнить файл с кодом как независимую программу. В обеих схемах могут быть использованы различные варианты, которые можно настроить по своему вкусу (в конце концов, это Python). Детали смотрите в методе `onRunCode` или просто отредактируйте и выполните свой собственный код Python.

На рис. 9.12 показаны четыре независимо выполняющиеся экземпляра PyEdit с различными цветовыми схемами, размерами и шрифтами. На этом рисунке видны также два оторванных меню PyEdit (в правом нижнем углу) и всплывающее окно PyEdit help (правый верхний угол). Фоном окон редактирования служат оттенки желтого, голубого, фиолетового и оранжевого; для установки желаемого цвета выберите в меню Tools элемент Pick.

Так как все эти четыре сеанса PyEdit редактируют исходный код Python, их содержимое можно выполнить через пункт Run Code выпадающего меню Tools. Код, запускаемый из файлов, выполняется независимым образом; стандартные потоки кода, выполняемого не из файла (например, полученные из самого графического элемента Text), отображаются в окно консоли сеанса PyEdit. Это никоим образом не является IDE (интегрированной средой разработки) – я добавил это только потому, что мне показалось это полезным. Очень удобно запускать редактируемый код, не разыскивая его в дереве каталогов.

Одно предостережение, прежде чем я натравлю вас на исходный код: в данной версии PyEdit еще нет кнопки «Отмена». Сам я не использую такого режима, и удаление и вставку можно легко отменить после того, как они сделаны (вставьте текст из буфера в прежнее место или удалите вставленный выделенный текст). Добавление обобщенной опции отмены действия составило бы хорошее упражнение для тех, кого это интересует. Возможный подход состоит в том, чтобы породить подклассы `TextEditor` или

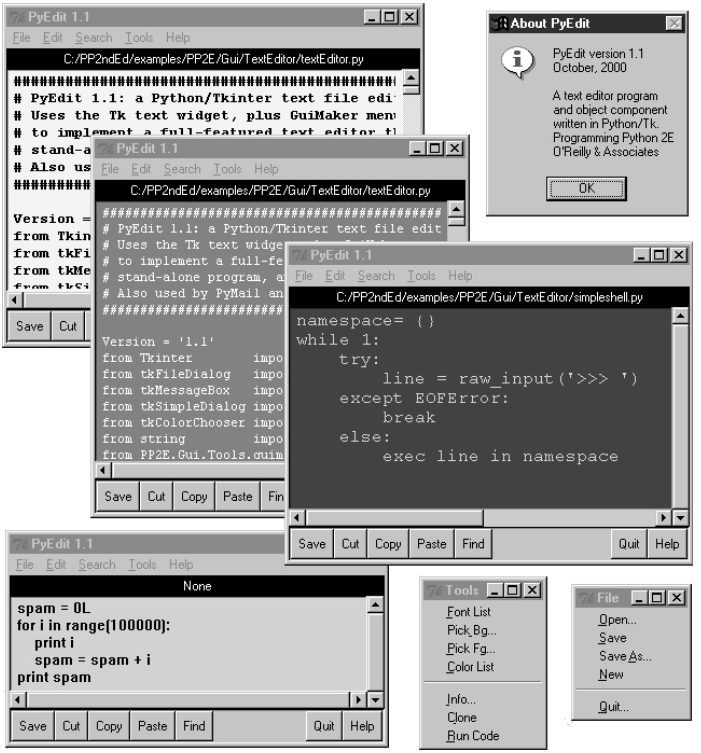


Рис. 9.12. Несколько одновременных сеансов PyEdit

самого класса Tkinter `Text`. Такой класс записывал бы операции с текстом в списке ограниченной длины и запускал вызовы для возвращения из каждой зарегистрированной операции по требованию и при обращении. С его помощью можно было бы также сделать PyEdit умнее, чтобы он знал, когда нужно запрашивать о сохранении файла перед выходом. В результате добавления «отмены» как внешнего класса существующий код PyEdit не потребовалось бы организовывать так, чтобы он отслеживал все, что делается с текстом. В конце концов, это Python.

Исходный код PyEdit

Программа в примере 9.14 состоит всего из двух исходных файлов – *.pyw*, который можно выполнять в Windows, чтобы избежать появления всплывающих окон консоли потоков DOS, и главного *.py*, который можно выполнять или импортировать. Нам требуются оба файла, так как PyEdit одновременно является сценарием и библиотекой, а файлы *.pyw* можно только запускать, но не импортировать (см. главу 2, если вы забыли, что это означает).

Пример 9.14. PP2E\Gui\TextEditor\textEditor.pyw

```
############################################################################
# выполнение PyEdit без окна консоли DOS для os.system в Windows;
# в настоящее время файлы ".pyw" нельзя импортировать как модули;
# если нужно, чтобы файл был программой, запускаемой без окна
# консоли dos в Windows, и мог импортироваться, используйте
```

```
# ".py" для главного файла и импортируйте код .py из ".pyw";
# execfile('textEditor.py') не работает при вызове из другого
# каталога, т. к. текущий каталог - тот, из которого я запущен;
########################################################################

import textEditor             # получить файл .py (или .pyc)
textEditor.main()             # запуск с точки входа верхнего уровня
```

Модуль примера 9.15 представляет собой реализацию PyEdit; главные классы для запуска и встраивания объекта PyEdit находятся в конце файла. Во время экспериментов с PyEdit изучайте этот листинг, чтобы разобраться в функциях и технологиях.

Пример 9.15. PP2E\Gui\TextEditor\textEditor.py

```
##############################################################################
# PyEdit 1.1: текстовый редактор и компонент на Python/Tkinter.
# Использует элемент текст из Tk, меню и кнопки GuiMaker для реализации
# полнофункционального редактора, который выполняется как самостоятельная
# программа или прикрепляется к другим GUI как компонент. Используется также
# в PyMail и PyView для редактирования почты и примечаний к графическому файлу.
##############################################################################

Version = '1.1'
from Tkinter        import *             # графические элементы, константы
from tkFileDialog   import *             # стандартные диалоги
from tkMessageBox   import *
from tkSimpleDialog import *
from tkColorChooser import askcolor
from string         import split, atoi
from PP2E.Gui.Tools.guimaker import *    # фрейм + построители меню/панелей инструментов
START     = '1.0'                        # индекс первого символа: row=1,col=0
SEL_FIRST = SEL + '.first'               # отобразить тег sel в индекс
SEL_LAST  = SEL + '.last'                # то же, что 'sel.last'

import sys, os, string
FontScale = 0                            # в Linux и других не-windows системах
if sys.platform[:3] != 'win':            # использовать более крупный шрифт
  FontScale = 3

class TextEditor:                   # смешать с классом фрейма, имеющим меню/панель инструментов
  startfiledir = '.'
  ftypes = [('All files',    '*'),             # для диалога открытия файла
        ('Text files',   '.txt'),             # настраивать в подклассе или
        ('Python files', '.py')]              # устанавливать в каждом экземпляре

  colors = [{'fg':'black',    'bg':'white'},  # список выбираемых цветов
        {'fg':'yellow',   'bg':'black'},      # первый элемент выбирается по умолчанию
        {'fg':'white',    'bg':'blue'},
        {'fg':'black',    'bg':'beige'},      # переделать по-своему или
        {'fg':'yellow',   'bg':'purple'},     # использовать выбор PickBg/Fg
        {'fg':'black',    'bg':'brown'},
        {'fg':'lightgreen', 'bg':'darkgreen'},
        {'fg':'darkblue',   'bg':'orange'},
        {'fg':'orange',   'bg':'darkblue'}]

  fonts  = [('courier',  9+FontScale, 'normal'), # нейтральные в отношении
        ('courier',   12+FontScale, 'normal'),   # платформы шрифты
        ('courier',   10+FontScale, 'bold'),     # (семейство,размер,стиль)
        ('courier',   10+FontScale, 'italic'),   # или вывод окна списка
        ('times',   10+FontScale, 'normal'),     # сделать крупнее в linux
```

```
      ('helvetica', 10+FontScale, 'normal'),
      ('arial',   10+FontScale, 'normal'),
      ('system',  10+FontScale, 'normal'),
      ('courier',   20+FontScale, 'normal')]

  def __init__(self, loadFirst=''):
    if not isinstance(self, GuiMaker):
      raise TypeError, 'TextEditor needs a GuiMaker mixin'
    self.setFileName(None)
    self.lastfind   = None
    self.openDialog = None
    self.saveDialog = None
    self.text.focus()              # иначе придется щелкнуть
    if loadFirst:
      self.onOpen(loadFirst)

  def start(self):                 # запускается из GuiMaker.__init__
    self.menuBar = [               # настройка меню/панели инструментов
      ('File', 0,
         [('Open...',  0, self.onOpen),
          ('Save',     0, self.onSave),
          ('Save As...', 5, self.onSaveAs),
          ('New',    0, self.onNew),
          'separator',
          ('Quit...',  0, self.onQuit)]
      ),
      ('Edit', 0,
         [('Cut',    0, self.onCut),
          ('Copy',     1, self.onCopy),
          ('Paste',    0, self.onPaste),
          'separator',
          ('Delete',   0, self.onDelete),
          ('Select All', 0, self.onSelectAll)]
      ),
      ('Search', 0,
         [('Goto...',  0, self.onGoto),
          ('Find...',  0, self.onFind),
          ('Refind',   0, self.onRefind),
          ('Change...',  0, self.onChange)]
      ),
      ('Tools', 0,
         [('Font List',   0, self.onFontList),
          ('Pick Bg...',  4, self.onPickBg),
          ('Pick Fg...',  0, self.onPickFg),
          ('Color List',  0, self.onColorList),
         'separator',
          ('Info...',  0, self.onInfo),
          ('Clone',    1, self.onClone),
          ('Run Code',   0, self.onRunCode)]
      )]
    self.toolBar = [
      ('Save',  self.onSave,   {'side': LEFT}),
      ('Cut',   self.onCut,  {'side': LEFT}),
      ('Copy',  self.onCopy,   {'side': LEFT}),
      ('Paste', self.onPaste,  {'side': LEFT}),
      ('Find',  self.onRefind, {'side': LEFT}),
      ('Help',  self.help,   {'side': RIGHT}),
      ('Quit',  self.onQuit,   {'side': RIGHT})]
```

```
def makeWidgets(self):                          # запускается из GuiMaker.__init__
  name = Label(self, bg='black', fg='white')  # добавить под меню и над панелью инструментов
  name.pack(side=TOP, fill=X)                 # меню/панели инструментов пакуются
  vbar  = Scrollbar(self)
  hbar  = Scrollbar(self, orient='horizontal')
  text  = Text(self, padx=5, wrap='none')

  vbar.pack(side=RIGHT,  fill=Y)
  hbar.pack(side=BOTTOM, fill=X)                # упаковать text последним
  text.pack(side=TOP,  fill=BOTH, expand=YES)  # иначе обрежутся полосы прокрутки
  text.config(yscrollcommand=vbar.set)         # вызывать vbar.set при перемещении по тексту
  text.config(xscrollcommand=hbar.set)
  vbar.config(command=text.yview)              # вызывать text.yview при прокрутке
  hbar.config(command=text.xview)              # или hbar['command']=text.xview

  text.config(font=self.fonts[0],
        bg=self.colors[0]['bg'], fg=self.colors[0]['fg'])
  self.text = text
  self.filelabel = name

######################
# Команды меню Edit
######################

def onCopy(self):                           # получить текст, выделенный мышью и т. п.
  if not self.text.tag_ranges(SEL):         # сохранить в общем буфере
    showerror('PyEdit', 'No text selected')
  else:
    text = self.text.get(SEL_FIRST, SEL_LAST)
    self.clipboard_clear()
    self.clipboard_append(text)

def onDelete(self):                         # удалить выделенный текст без сохранения
  if not self.text.tag_ranges(SEL):
    showerror('PyEdit', 'No text selected')
  else:
    self.text.delete(SEL_FIRST, SEL_LAST)

def onCut(self):
  if not self.text.tag_ranges(SEL):
    showerror('PyEdit', 'No text selected')
  else:
    self.onCopy()                           # сохранить и удалить выделенный текст
    self.onDelete()

def onPaste(self):
  try:
    text = self.selection_get(selection='CLIPBOARD')
  except TclError:
    showerror('PyEdit', 'Nothing to paste')
    return
  self.text.insert(INSERT, text)            # добавить в текущем положении курсора
  self.text.tag_remove(SEL, '1.0', END)
  self.text.tag_add(SEL, INSERT+'-%dc' % len(text), INSERT)
  self.text.see(INSERT)                     # выделить, чтобы можно было удалить

def onSelectAll(self):
  self.text.tag_add(SEL, '1.0', END+'-1c')  # выделить весь текст
  self.text.mark_set(INSERT, '1.0')         # переместить точку вставки вверх
  self.text.see(INSERT)                     # прокрутка до верха
```

```
######################
# Команды меню Tools
######################

def onFontList(self):
  self.fonts.append(self.fonts[0])          # выбрать следующий шрифт из списка
  del self.fonts[0]                         # изменить размер текстовой области
  self.text.config(font=self.fonts[0])

def onColorList(self):
  self.colors.append(self.colors[0])        # выбрать следующий цвет из списка
  del self.colors[0]                        # текущий сместить в конец
  self.text.config(fg=self.colors[0]['fg'], bg=self.colors[0]['bg'])

def onPickFg(self):
  self.pickColor('fg')                      # добавлено 10/02/00
def onPickBg(self):                         # выбрать произвольный цвет
  self.pickColor('bg')                      # в стандартном диалоге цвета
def pickColor(self, part):                  # это очень просто
  (triple, hexstr) = askcolor()
  if hexstr:
    apply(self.text.config, (), {part: hexstr})

def onInfo(self):
  text  = self.getAllText()                 # добавлено 5.3.00 за 15 мин.
  bytes = len(text)                         # за words принимается все, что
  lines = len(string.split(text, '\n'))     # разделяется пробельными символами
  words = len(string.split(text))
  index = self.text.index(INSERT)
  where = tuple(string.split(index, '.'))
  showinfo('PyEdit Information',
       'Current location:\n\n' +
       'line:\t%s\ncolumn:\t%s\n\n' % where +
       'File text statistics:\n\n' +
       'bytes:\t%d\nlines:\t%d\nwords:\t%d\n' % (bytes, lines, words))

def onClone(self):
  new = Toplevel()                    # новое окно редактора в том же процессе
  myclass = self.__class__            # объект класса экземпляра (самый нижний)
  myclass(new)                        # прикрепить/выполнить экземпляр моего класса

def onRunCode(self, parallelmode=1):
  """
  выполнение редактируемого кода Python - это не IDE, но удобно;
  пытается работать в каталоге файла, а не cwd (может быть корнем pp2e);
  вводит и добавляет аргументы командной строки для файлов сценариев;
  для кода stdin/out/err = стартовому окну редактора, если оно есть;
  но parallelmode открывает окно dos для i/o;
  """
  from PP2E.launchmodes import System, Start, Fork
  filemode = 0
  thefile  = str(self.getFileName())
  cmdargs  = askstring('PyEdit', 'Commandline arguments?') or ''
  if os.path.exists(thefile):
    filemode = askyesno('PyEdit', 'Run from file?')
  if not filemode:                                  # выполнение как строки текста
    namespace = {'__name__': '__main__'}            # выполнение на верхнем уровне
    sys.argv = [thefile] + string.split(cmdargs)    # можно использовать потоки
    exec self.getAllText() + '\n' in namespace      # исключительные ситуации игнорируются
  elif askyesno('PyEdit', 'Text saved in file?'):
```

```
    mycwd = os.getcwd()                                 # cwd может быть корнем
    os.chdir(os.path.dirname(thefile) or mycwd)         # cd по имени файла
    thecmd  = thefile + ' ' + cmdargs
    if not parallelmode:                                # выполнение как файла
      System(thecmd, thecmd)()                          # блокировать редактор editor
    else:
      if sys.platform[:3] == 'win':                     # породить параллельно
        Start(thecmd, thecmd)()                         # или с помощью os.spawnv
      else:
        Fork(thecmd, thecmd)()                          # породить параллельно
    os.chdir(mycwd)

########################
# Команды меню Search
########################

def onGoto(self):
  line = askinteger('PyEdit', 'Enter line number')
  self.text.update()
  self.text.focus()
  if line is not None:
    maxindex = self.text.index(END+'-1c')
    maxline  = atoi(split(maxindex, '.')[0])
    if line > 0 and line <= maxline:
      self.text.mark_set(INSERT, '%d.0' % line)         # идти на строку
      self.text.tag_remove(SEL, '1.0', END)             # удалить выделенное
      self.text.tag_add(SEL, INSERT, 'insert + 1l')     # выделить строку
      self.text.see(INSERT)                             # прокрутка до строки
    else:
      showerror('PyEdit', 'Bad line number')

def onFind(self, lastkey=None):
  key = lastkey or askstring('PyEdit', 'Enter search string')
  self.text.update()
  self.text.focus()
  self.lastfind = key
  if key:
    where = self.text.search(key, INSERT, END)    # не переносить в начало
    if not where:
      showerror('PyEdit', 'String not found')
    else:
      pastkey = where + '+%dc' % len(key)           # индекс после ключа
      self.text.tag_remove(SEL, '1.0', END)         # удалить выделения
      self.text.tag_add(SEL, where, pastkey)        # выделить ключ
      self.text.mark_set(INSERT, pastkey)           # для следующего поиска
      self.text.see(where)                          # прокрутить экран

def onRefind(self):
  self.onFind(self.lastfind)

def onChange(self):
  new = Toplevel(self)
  Label(new, text='Find text:').grid(row=0, column=0)
  Label(new, text='Change to:').grid(row=1, column=0)
  self.change1 = Entry(new)
  self.change2 = Entry(new)
  self.change1.grid(row=0, column=1, sticky=EW)
  self.change2.grid(row=1, column=1, sticky=EW)
  Button(new, text='Find',
```

```
        command=self.onDoFind).grid(row=0, column=2, sticky=EW)
    Button(new, text='Apply',
        command=self.onDoChange).grid(row=1, column=2, sticky=EW)
    new.columnconfigure(1, weight=1)                # расширяемые поля ввода

  def onDoFind(self):
    self.onFind(self.change1.get())                 # найти то, что в окне замены

  def onDoChange(self):
    if self.text.tag_ranges(SEL):                   # сначала найти
      self.text.delete(SEL_FIRST, SEL_LAST)         # применить замену
      self.text.insert(INSERT, self.change2.get())  # удаляет, если пусто
      self.text.see(INSERT)
      self.onFind(self.change1.get())               # переход к следующему
      self.text.update()                            # заставить обновиться

  ######################
  # Команды меню File
  ######################

  def my_askopenfilename(self):       # объекты помнят каталог/файл последнего результата
    if not self.openDialog:
       self.openDialog = Open(initialdir=self.startfiledir,
                   filetypes=self.ftypes)
    return self.openDialog.show()

  def my_asksaveasfilename(self):     # объекты помнят каталог/файл последнего результата
    if not self.saveDialog:
       self.saveDialog = SaveAs(initialdir=self.startfiledir,
                   filetypes=self.ftypes)
    return self.saveDialog.show()

  def onOpen(self, loadFirst=''):
    doit = self.isEmpty() or askyesno('PyEdit', 'Disgard text?')
    if doit:
      file = loadFirst or self.my_askopenfilename()
      if file:
        try:
          text = open(file, 'r').read()
        except:
          showerror('PyEdit', 'Could not open file ' + file)
        else:
          self.setAllText(text)
          self.setFileName(file)

  def onSave(self):
    self.onSaveAs(self.currfile)     # может оказаться None

  def onSaveAs(self, forcefile=None):
    file = forcefile or self.my_asksaveasfilename()
    if file:
      text = self.getAllText()
      try:
        open(file, 'w').write(text)
      except:
        showerror('PyEdit', 'Could not write file ' + file)
      else:
        self.setFileName(file)       # может быть заново создаваемым

  def onNew(self):
    doit = self.isEmpty() or askyesno('PyEdit', 'Disgard text?')
```

```
    if doit:
      self.setFileName(None)
      self.clearAllText()

  def onQuit(self):
    if askyesno('PyEdit', 'Really quit PyEdit?'):
      self.quit()                  # Frame.quit через GuiMaker

  ####################################
  # Прочие, полезные вне этого класса
  ####################################

  def isEmpty(self):
    return not self.getAllText()

  def getAllText(self):
    return self.text.get('1.0', END+'-1c')  # извлечь текст как строку

  def setAllText(self, text):
    self.text.delete('1.0', END)      # записать текстовую строку в элемент
    self.text.insert(END, text)       # или '1.0'
    self.text.mark_set(INSERT, '1.0') # переместить точку ввода в начало
    self.text.see(INSERT)             # прокрутка туда, где точка вставки

  def clearAllText(self):
    self.text.delete('1.0', END)      # удалить текст элемента

  def getFileName(self):
    return self.currfile

  def setFileName(self, name):
    self.currfile = name  # for save
    self.filelabel.config(text=str(name))

  def help(self):
    showinfo('About PyEdit',
         'PyEdit version %s\nOctober, 2000\n\n'
         'A text editor program\nand object component\n'
         'written in Python/Tk.\nProgramming Python 2E\n'
         "O'Reilly & Associates" % Version)

##################################################################
# готовые к употреблению классы редактора соединить
# с подклассом Frame, который строит меню/панели инструментов
##################################################################

# когда редактор является владельцем окна
class TextEditorMain(TextEditor, GuiMakerWindowMenu):    # добавить конструктор
                                                         # меню/панели инструментов
  def __init__(self, parent=None, loadFirst=''):         # когда заполняет все окно
    GuiMaker.__init__(self, parent)                      # использовать меню главного окна
    TextEditor.__init__(self, loadFirst)                 # у self есть фрейм GuiMaker
    self.master.title('PyEdit ' + Version)               # заголовок в автономной версии
    self.master.iconname('PyEdit')                       # перехват кнопки wm delete
    self.master.protocol('WM_DELETE_WINDOW', self.onQuit)

class TextEditorMainPopup(TextEditor, GuiMakerWindowMenu):
  def __init__(self, parent=None, loadFirst=''):
    self.popup = Toplevel(parent)                        # создать свое окно
    GuiMaker.__init__(self, self.popup)                  # использовать меню главного окна
    TextEditor.__init__(self, loadFirst)
    assert self.master == self.popup
```

```
    self.popup.title('PyEdit ' + Version)
    self.popup.iconname('PyEdit')
  def quit(self):
    self.popup.destroy()                                  # убить только это окно

# если встраивается в другое окно
class TextEditorComponent(TextEditor, GuiMakerFrameMenu):
  def __init__(self, parent=None, loadFirst=''):          # меню на основе Frame
    GuiMaker.__init__(self, parent)                       # все меню, кнопки
    TextEditor.__init__(self, loadFirst)                  # GuiMaker инициируется первым

class TextEditorComponentMinimal(TextEditor, GuiMakerFrameMenu):
  def __init__(self, parent=None, loadFirst='', deleteFile=1):
    self.deleteFile = deleteFile
    GuiMaker.__init__(self, parent)
    TextEditor.__init__(self, loadFirst)
  def start(self):
    TextEditor.start(self)                                # вызов запуска GuiMaker
    for i in range(len(self.toolBar)):                    # убрать quit из панели инструментов
      if self.toolBar[i][0] == 'Quit':                    # убрать пункты меню file
        del self.toolBar[i]; break                        # или просто отключить
    if self.deleteFile:
      for i in range(len(self.menuBar)):
        if self.menuBar[i][0] == 'File':
          del self.menuBar[i]; break
    else:
      for (name, key, items) in self.menuBar:
        if name == 'File':
          items.append([1,2,3,4,6])

# выполнение самостоятельной программы
def testPopup():
  # см. проверку компонент в PyView и PyMail
  root = Tk()
  TextEditorMainPopup(root)
  TextEditorMainPopup(root)
  Button(root, text='More', command=TextEditorMainPopup).pack(fill=X)
  Button(root, text='Quit', command=root.quit).pack(fill=X)
  root.mainloop()

def main():                             # можно ввести с клавиатуры или щелкнуть
  try:                                  # либо ассоциировано имя файла в Windows
    fname = sys.argv[1]                 # arg = необязательное имя файла
  except IndexError:
    fname = None
  TextEditorMain(loadFirst=fname).pack(expand=YES, fill=BOTH)
  mainloop()

if __name__ == '__main__':              # если выполняется как сценарий
  #testPopup()
  main()                                # выполнять .pyw, чтобы не было окна dos
```

PyView: слайд-шоу для графики и заметок

Лучше один раз увидеть, чем потратить тысячу слов, но их понадобится значительно меньше, чтобы вывести картинку с помощью Python. В следующей программе, PyView, реализован простой алгоритм слайд-шоу с помощью переносимого кода Python/Tkinter.

Выполнение PyView

В PyView соединились многие из тем, изучавшихся в последней главе: последовательность показа регулируется событиями `after`, графические объекты выводятся на холсте, размер которого автоматически изменяется, и т. д. В главном окне программы на холсте выводится фотография; пользователь может открыть и просматривать ее непосредственно или запустить режим показа слайдов, в котором выводятся фотографии, случайным образом выбранные из каталога, через регулярные промежутки времени, задаваемые с помощью ползунка.

По умолчанию показ слайдов PyView производится для каталога с графикой с прилагаемого CD (хотя кнопка Open позволяет загружать графику из любых каталогов). Чтобы посмотреть другую группу фотографий, передайте имя каталога в качестве первого аргумента командной строки или измените имя каталога по умолчанию в самом сценарии. Показ слайдов здесь не воспроизвести, но главное окно привести можно. На рис. 9.13 изображено главное окно PyView по умолчанию.

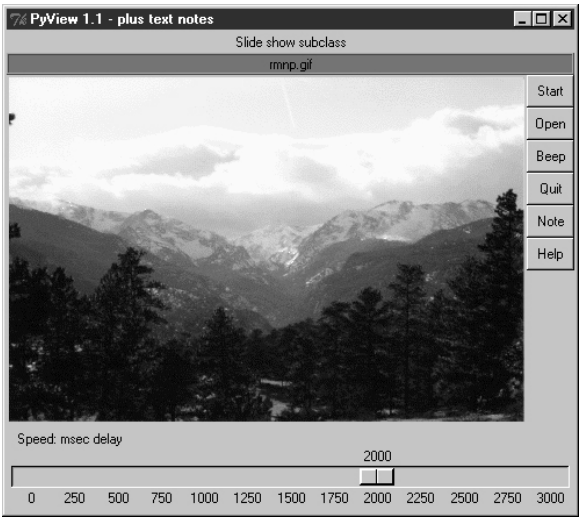


Рис. 9.13. PyView без текста

На рисунке этого не видно, но в действительности на метке вверху окна черным по красному выведен путь к отображаемому файлу. Сейчас переместите ползунок до конца к отметке «0», что задает отсутствие задержки между сменой фотографий, и щелкните по кнопке Start, чтобы начать очень быстрый показ слайдов. Если ваш компьютер обладает хотя бы таким же быстродействием, как мой, то фотографии будут мелькать слишком быстро, чтобы их можно было где-либо применить, кроме как в рекламе, действующей на подсознание. Демонстрируемые фотографии загружаются при начальном запуске, чтобы сохранить ссылки на них (помните, что объекты с изображениями нужно удерживать). Но скорость, с которой в Python большие GIF-файлы выводятся в окно, впечатляет, а то и просто восхищает.

Во время показа слайдов кнопка GUI Start изменяется на Stop (изменяется ее атрибут текста с помощью метода элемента `config`). Рис. 9.14 показывает, что находится на экране после нажатия в подходящий момент кнопки Stop.

Кроме того, у каждой фотографии может быть свой файл «примечаний», который автоматически открывается вместе с изображением. С помощью этой функции можно

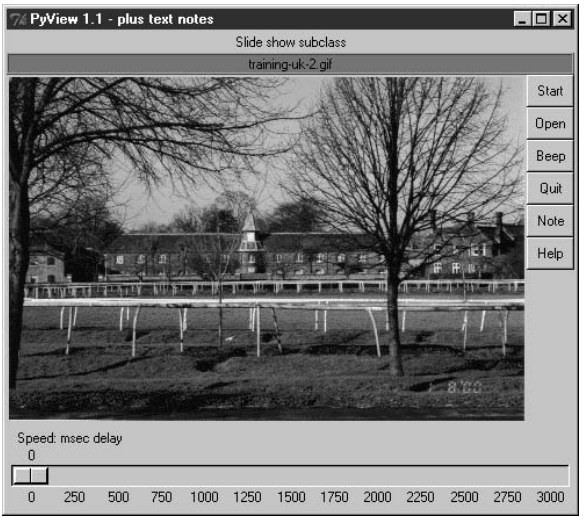


Рис. 9.14. PyView после остановки показа слайдов

записывать основные данные о фотографии. Нажмите кнопку Note, чтобы открыть дополнительный набор графических элементов, с помощью которых можно просматривать и изменять файл примечаний, связанный с просматриваемой в данный момент фотографией. Этот дополнительный набор элементов должен показаться вам знакомым – это текстовый редактор PyEdit из предыдущего раздела, прикрепленный к PyView в качестве средства просмотра и редактирования примечаний к фотографиям. На рис. 9.15 показан PyView вместе с открытым прикрепленным к нему компонентом PyEdit для редактирования примечаний.

В результате получается очень большое окно, которое обычно лучше просматривать развернутым на весь экран. Однако главное, на что нужно обратить внимание, это правый нижний угол экрана над ползунком – там находится прикрепленный объект PyEdit, выполняющий тот же самый код, который приведен в предыдущем разделе. Так как PyEdit реализован в виде класса GUI, подобным же образом можно повторно использовать его в любом GUI, которому требуется интерфейс для редактирования текста. При встраивании таким способом меню PyEdit основываются на фрейме (он не владеет окном в целом), текстовое содержимое запоминается и выбирается непосредственно, а некоторые возможности автономного режима опущены (например, не стало выпадающего меню File).

Средство просмотра примечаний появляется только при нажатии кнопки Note и удаляется при повторном ее нажатии; для того чтобы показать или скрыть фрейм просмотра примечаний, PyView пользуется методами графических элементов `pack` и `pack_forget`, с которыми мы познакомились в конце предыдущей главы. Окно автоматически расширяется, чтобы разместить средство просмотра примечаний, когда оно пакуется и отображается. Можно открыть файл примечаний во всплывающем окне PyEdit, но PyView встраивает редактор, чтобы сохранить прямую зрительную ассоциацию. Мы еще встретимся с таким встраиванием PyEdit внутри другого GUI, когда будем рассматривать PyMail в главе 11.

Предупреждение: в таком виде PyView поддерживает те же графические форматы, что и объект `PhotoImage` библиотеки Tkinter, поэтому по умолчанию он ищет файлы GIF. Улучшить положение можно, установив расширение PIL для просмотра JPEG

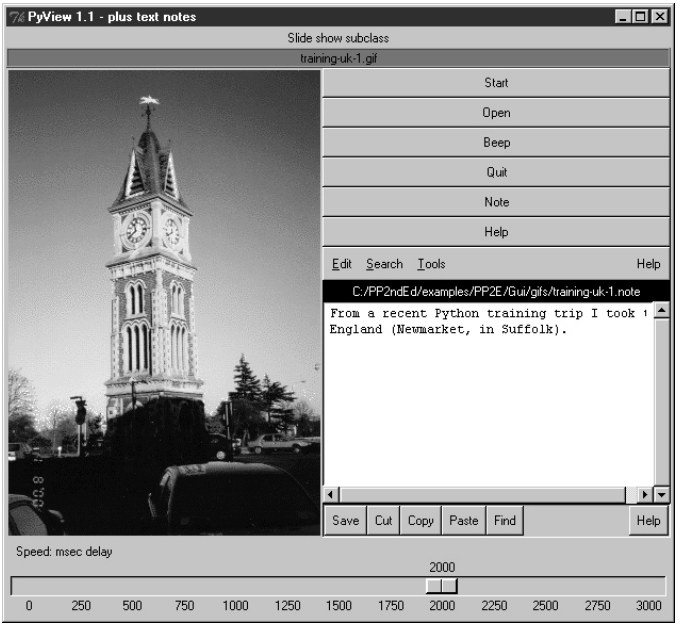


Рис. 9.15. PyView с примечаниями

(и многих других форматов). Поскольку сегодня PIL является факультативным расширением, он не включен в данную версию PyView. Подробнее о PIL и графических форматах сказано в конце главы 7.

Исходный код PyView

Поскольку программа PyView реализовывалась поэтапно, чтобы понять, как она в действительности работает, нужно изучить объединение, состоящее из двух файлов и классов. В одном файле реализован класс, предоставляющий основные функции показа слайдов, а в другом реализован класс, расширяющий исходный и добавляющий новые функции поверх базового поведения. Начнем с класса расширения: пример 9.16 добавляет ряд функций в импортируемый базовый класс показа слайдов – редактирование примечаний, ползунок для задания задержки, метка файла и т. д. Это тот файл, который фактически выполняется для запуска PyView.

Пример 9.16. PP2E\Gui\SlideShow\slideShowPlus.py

```
##############################################################################################
# SlideShowPlus: добавить файлы примечаний с помощью прикрепленного объекта PyEdit, ползунок
# для установки интервала задержки показа и метку с именем отображаемого в данный момент файла;
##############################################################################################

import os, string
from Tkinter import *
from PP2E.Gui.TextEditor.textEditor import *
from slideShow import SlideShow
#from slideShow_threads import SlideShow

class SlideShowPlus(SlideShow):
    def __init__(self, parent, picdir, editclass, msecs=2000):
```

```
        self.msecs = msecs
        self.editclass = editclass
        SlideShow.__init__(self, parent=parent, picdir=picdir, msecs=msecs)
    def makeWidgets(self):
        self.name = Label(self, text='None', bg='red', relief=RIDGE)
        self.name.pack(fill=X)
        SlideShow.makeWidgets(self)
        Button(self, text='Note', command=self.onNote).pack(fill=X)
        Button(self, text='Help', command=self.onHelp).pack(fill=X)
        s = Scale(label='Speed: msec delay', command=self.onScale,
                  from_=0, to=3000, resolution=50, showvalue=YES,
                  length=400, tickinterval=250, orient='horizontal')
        s.pack(side=BOTTOM, fill=X)
        s.set(self.msecs)
        if self.editclass == TextEditorMain:                # теперь сделать редактор
            self.editor = self.editclass(self.master)       # нужен корень для меню
        else:
            self.editor = self.editclass(self)              # встроенный или всплывающий
        self.editor.pack_forget()                           # первоначально скрыть редактор
        self.editorUp = self.image = None
    def onStart(self):
        SlideShow.onStart(self)
        self.config(cursor='watch')
    def onStop(self):
        SlideShow.onStop(self)
        self.config(cursor='hand2')
    def onOpen(self):
        SlideShow.onOpen(self)
        if self.image:
            self.name.config(text=os.path.split(self.image[0])[1])
        self.config(cursor='crosshair')
        self.switchNote()
    def quit(self):
        self.saveNote()
        SlideShow.quit(self)
    def drawNext(self):
        SlideShow.drawNext(self)
        if self.image:
            self.name.config(text=os.path.split(self.image[0])[1])
        self.loadNote()
    def onScale(self, value):
        self.msecs = string.atoi(value)
    def onNote(self):
        if self.editorUp:                   # если редактор уже открыт
            #self.saveNote()                # сохранить текст, скрыть редактор
            self.editor.pack_forget()
            self.editorUp = 0
        else:
            self.editor.pack(side=TOP)      # либо показать/упаковать редактор
            self.editorUp = 1               # и загрузить текст примечания к фото
            self.loadNote()
    def switchNote(self):
        if self.editorUp:
            self.saveNote()         # сохранить примечание к тексту текущего изображения
            self.loadNote()         # загрузить примечание для нового изображения
    def saveNote(self):
        if self.editorUp:
            currfile = self.editor.getFileName()     # или self.editor.onSave()
```

```
            currtext = self.editor.getAllText()      # но текст может быть пуст
            if currfile and currtext:
                try:
                    open(currfile, 'w').write(currtext)
                except:
                    pass # нормально, если запускается не с cd
    def loadNote(self):
        if self.image and self.editorUp:
            root, ext = os.path.splitext(self.image[0])
            notefile  = root + '.note'
            self.editor.setFileName(notefile)
            try:
                self.editor.setAllText(open(notefile).read())
            except:
                self.editor.clearAllText()
    def onHelp(self):
        showinfo('About PyView',
                 'PyView version 1.1\nJuly, 1999\n'
                 'An image slide show\nProgramming Python 2E')

if __name__ == '__main__':
    import sys
    picdir = '../gifs'
    if len(sys.argv) >= 2:
        picdir = sys.argv[1]

    editstyle = TextEditorComponentMinimal
    if len(sys.argv) == 3:
        try:
            editstyle = [TextEditorMain,
                         TextEditorMainPopup,
                         TextEditorComponent,
                         TextEditorComponentMinimal][string.atoi(sys.argv[2])]
        except: pass

    root = Tk()
    root.title('PyView 1.1 - plus text notes')
    Label(root, text="Slide show subclass").pack()
    SlideShowPlus(parent=root, picdir=picdir, editclass=editstyle)
    root.mainloop()
```

Базовые функции, расширяемые `SlideShowPlus`**, находятся в примере 9.17. Это было первоначальной реализацией показа слайдов; она открывает изображения, выводит изображения и организует показ слайдов. Можно запустить ее самостоятельно, но не будет получено таких развитых функций, как примечания и ползунки, добавляемые в подклассе** `SlideShowPlus`**.**

Пример 9.17. PP2E\Gui\SlideShow\slideShow.py

```
##############################################################
# SlideShow: простой показ слайдов на Python/Tkinter;
# базовый набор функций можно расширять в подклассах;
##############################################################

from Tkinter import *
from glob import glob
from tkMessageBox import askyesno
from tkFileDialog import askopenfilename
import random
Width, Height = 450, 450
```

```
imageTypes = [('Gif files', '.gif'),    # для диалога открытия файла
              ('Ppm files', '.ppm'),    # плюс jpg с патчем Tk,
              ('Pgm files', '.pgm'),    # плюс растровые с помощью BitmapImage
              ('All files', '*')]

class SlideShow(Frame):
    def __init__(self, parent=None, picdir='.', msecs=3000, **args):
        Frame.__init__(self, parent, args)
        self.makeWidgets()
        self.pack(expand=YES, fill=BOTH)
        self.opens = picdir
        files = []
        for label, ext in imageTypes[:-1]:
            files = files + glob('%s/*%s' % (picdir, ext))
        self.images = map(lambda x: (x, PhotoImage(file=x)), files)
        self.msecs  = msecs
        self.beep   = 1
        self.drawn  = None
    def makeWidgets(self):
        self.canvas = Canvas(self, bg='white', height=Height, width=Width)
        self.canvas.pack(side=LEFT, fill=BOTH, expand=YES)
        self.onoff = Button(self, text='Start', command=self.onStart)
        self.onoff.pack(fill=X)
        Button(self, text='Open',  command=self.onOpen).pack(fill=X)
        Button(self, text='Beep',  command=self.onBeep).pack(fill=X)
        Button(self, text='Quit',  command=self.onQuit).pack(fill=X)
    def onStart(self):
        self.loop = 1
        self.onoff.config(text='Stop', command=self.onStop)
        self.canvas.config(height=Height, width=Width)
        self.onTimer()
    def onStop(self):
        self.loop = 0
        self.onoff.config(text='Start', command=self.onStart)
    def onOpen(self):
        self.onStop()
        name = askopenfilename(initialdir=self.opens, filetypes=imageTypes)
        if name:
            if self.drawn: self.canvas.delete(self.drawn)
            img = PhotoImage(file=name)
            self.canvas.config(height=img.height(), width=img.width())
            self.drawn = self.canvas.create_image(2, 2, image=img, anchor=NW)
            self.image = name, img
    def onQuit(self):
        self.onStop()
        self.update()
        if askyesno('PyView', 'Really quit now?'):
            self.quit()
    def onBeep(self):
        self.beep = self.beep ^ 1
    def onTimer(self):
        if self.loop:
            self.drawNext()
            self.after(self.msecs, self.onTimer)
    def drawNext(self):
        if self.drawn: self.canvas.delete(self.drawn)
        name, img  = random.choice(self.images)
        self.drawn = self.canvas.create_image(2, 2, image=img, anchor=NW)
```

```
            self.image = name, img
            if self.beep: self.bell()
            self.canvas.update()

    if __name__ == '__main__':
        import sys
        if len(sys.argv) == 2:
            picdir = sys.argv[1]
        else:
            picdir = '../gifs'
        root = Tk()
        root.title('PyView 1.0')
        root.iconname('PyView')
        Label(root, text="Python Slide Show Viewer").pack()
        SlideShow(root, picdir=picdir, bd=3, relief=SUNKEN)
        root.mainloop()
```

Чтобы дать лучшее представление о том, что реализовано этим базовым классом, на рис. 9.16 показано, как он выглядит при самостоятельном выполнении (в действительности это два экземпляра, выполняемые самостоятельно) сценарием slideShow_frames, который находится на прилагаемом CD.

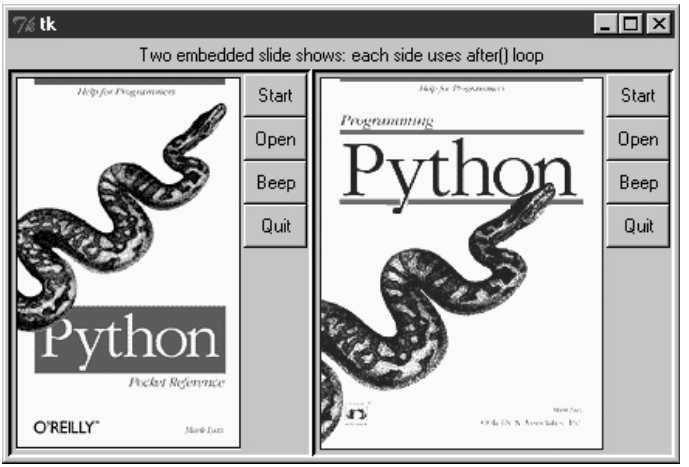


Рис. 9.16. Два прикрепленных объекта SlideShow

Простой сценарий slideShow_frames прикрепляет два экземпляра SlideShow к одному окну, что возможно благодаря записи информации о состоянии в переменные экземпляра класса, а не в глобальные переменные. Сценарий slideShow_toplevels, также имеющийся на CD, прикрепляет два SlideShow к двум всплывающим окнам верхнего уровня. В обоих случаях показ слайдов происходит независимо, но управляется событиями after, генерируемыми одним и тем же циклом событий в одном процессе.

PyDraw: рисование и перемещение графики

В предыдущей главе мы познакомились с простой техникой анимации в Tkinter (см. версии canvasDraw в обзоре). Приводимая здесь программа PyDraw, основываясь на тех же идеях, реализует на Python более богатые функциональные возможности. В ней появились новые режимы рисования trails и scribble, заливка объекта и фона,

встраивание фотографий и другое. Кроме того, реализованы приемы перемещения объектов и анимации – нарисованные объекты можно перемещать по холсту щелчками и перетаскиванием, а любой нарисованный объект можно плавно переместить через экран в место, указанное щелчком мыши.

Выполнение PyDraw

PyDraw, по существу, представляет собой холст Tkinter с многочисленными привязками событий клавиатуры и мыши, которые дают возможность пользователю осуществлять стандартные операции рисования. Ни по каким меркам это нельзя назвать графической программой профессионального уровня, но поразвлечься с ней можно. На самом деле даже нужно, поскольку такой носитель, как книга, не позволяет передать такие вещи, как движущийся объект. Запустите PyDraw из какой-либо панели запуска программ (или прямо файл *movingpics.py* из примера 9.18). Нажмите клавишу <?> и посмотрите подсказку по всем имеющимся командам (или прочтите строку help в листинге).

На рис. 9.17 показано окно PyDraw после изображения на холсте нескольких объектов. Чтобы переместить какой-либо из показанных объектов, щелкните по нему средней кнопкой мыши и перетаскивайте курсором либо щелкните средней кнопкой по объекту, а затем правой кнопкой в том месте, куда вы хотите его переместить. В последнем случае PyDraw осуществляет анимацию, постепенно перемещая объект в указанное место. Попробуйте сделать это с находящейся вверху фотографией создателя Python Гвидо ван Россума и вы увидите известный демонстрационный ролик «Движущийся Гвидо» (не сомневайтесь, у него тоже есть чувство юмора).

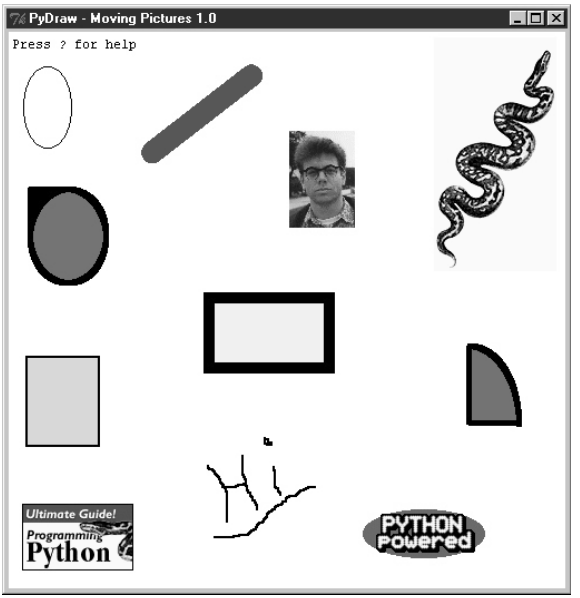


Рис. 9.17. PyDraw с рисованными объектами, готовыми двинуться

Для вставки фотографий нажмите <P>, для рисования фигур нажмите левую кнопку и вытягивайте их. Пользователям Windows: щелчок средней кнопкой обычно равносилен нажатию двух кнопок одновременно, но может потребоваться настроить это в панели управления. Помимо событий мыши есть еще 17 команд клавиш для редакти-

рования рисунков, но о них здесь не будет рассказано. Требуется некоторое время, чтобы освоиться со всеми командами клавиатуры и мыши, после чего вы тоже сможете создавать бессмысленные электронные художества, как на рис. 9.18.

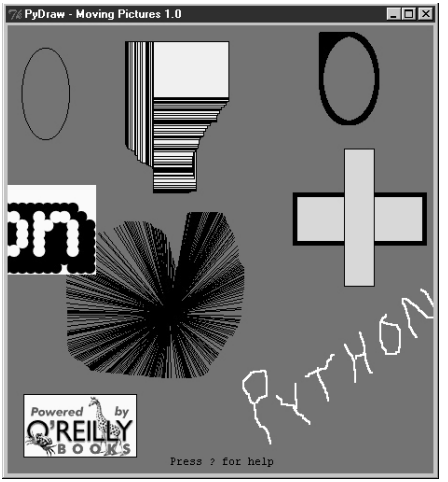


Рис. 9.18. PyDraw после некоторых игр с ним

Исходный код PyDraw

Как и PyEdit, PyDraw размещается в одном файле. За главным модулем, показанным в примере 9.18, приведены два расширения, изменяющие реализацию перемещения.

Пример 9.18. PP2E\Gui\MovingPics\movingpics.py

```
############################################################################
# PyDraw: простая программа рисования на холсте и перемещения/анимации объектов
# перемещение объектов осуществляется в циклах time.sleep, поэтому одновременно
# может происходить только одно такое перемещение; оно плавное и быстрое,
# но смотрите здесь же другие приемы с использованием .after и потоков;
############################################################################

helpstr = """--PyDraw version 1.0--
Mouse commands:
  Left        = Set target spot
  Left+Move   = Draw new object
  Double Left = Clear all objects
  Right       = Move current object
  Middle      = Select closest object
  Middle+Move = Drag current object

Keyboard commands:
  w=Pick border width  c=Pick color
  u=Pick move unit     s=Pick move delay
  o=Draw ovals         r=Draw rectangles
  l=Draw lines         a=Draw arcs
  d=Delete object      1=Raise object
  2=Lower object       f=Fill object
  b=Fill background    p=Add photo
  z=Save postscript    x=Pick pen modes
```

```
  ?=Help               other=clear text
"""

import time, sys
from Tkinter import *
from tkFileDialog import *
from tkMessageBox import *
PicDir = '../gifs'

if sys.platform[:3] == 'win':
    HelpFont = ('courier', 9, 'normal')
else:
    HelpFont = ('courier', 12, 'normal')

pickDelays = [0.01, 0.025, 0.05, 0.10, 0.25, 0.0, 0.001, 0.005]
pickUnits  = [1, 2, 4, 6, 8, 10, 12]
pickWidths = [1, 2, 5, 10, 20]
pickFills  = [None,'white','blue','red','black','yellow','green','purple']
pickPens   = ['elastic', 'scribble', 'trails']

class MovingPics:
    def __init__(self, parent=None):
        canvas = Canvas(parent, width=500, height=500, bg= 'white')
        canvas.pack(expand=YES, fill=BOTH)
        canvas.bind('<ButtonPress-1>',  self.onStart)
        canvas.bind('<B1-Motion>',      self.onGrow)
        canvas.bind('<Double-1>',       self.onClear)
        canvas.bind('<ButtonPress-3>',  self.onMove)
        canvas.bind('<Button-2>',       self.onSelect)
        canvas.bind('<B2-Motion>',      self.onDrag)
        parent.bind('<KeyPress>',       self.onOptions)
        self.createMethod = Canvas.create_oval
        self.canvas = canvas
        self.moving = []
        self.images = []
        self.object = None
        self.where  = None
        self.scribbleMode = 0
        parent.title('PyDraw - Moving Pictures 1.0')
        parent.protocol('WM_DELETE_WINDOW', self.onQuit)
        self.realquit = parent.quit
        self.textInfo = self.canvas.create_text(
                                5, 5, anchor=NW,
                                font=HelpFont,
                                text='Press ? for help')
    def onStart(self, event):
        self.where  = event
        self.object = None
    def onGrow(self, event):
        canvas = event.widget
        if self.object and pickPens[0] == 'elastic':
            canvas.delete(self.object)
        self.object = self.createMethod(canvas,
                               self.where.x, self.where.y,    # начало
                               event.x, event.y,              # конец
                               fill=pickFills[0], width=pickWidths[0])
        if pickPens[0] == 'scribble':
            self.where = event              # следующий раз отсюда
    def onClear(self, event):
```

```
        if self.moving: return              # ok, если движется
        event.widget.delete('all')          # с помощью тега all
        self.images = []
        self.textInfo = self.canvas.create_text(
                                5, 5, anchor=NW,
                                font=HelpFont,
                                text='Press ? for help')
    def plotMoves(self, event):
        diffX  = event.x - self.where.x             # планирование перемещений анимации
        diffY  = event.y - self.where.y             # по горизонтали, затем по вертикали
        reptX  = abs(diffX) / pickUnits[0]          # приращение в каждом шаге, число шагов
        reptY  = abs(diffY) / pickUnits[0]          # от предыдущего до щелчка
        incrX  = pickUnits[0] * ((diffX > 0) or -1)
        incrY  = pickUnits[0] * ((diffY > 0) or -1)
        return incrX, reptX, incrY, reptY
    def onMove(self, event):
        traceEvent('onMove', event, 0)      # переместить текущий объект в место щелчка
        object = self.object                # игнорировать во время движения некоторые операции
        if object and object not in self.moving:
            msecs  = int(pickDelays[0] * 1000)
            parms  = 'Delay=%d msec, Units=%d' % (msecs, pickUnits[0])
            self.setTextInfo(parms)
            self.moving.append(object)
            canvas = event.widget
            incrX, reptX, incrY, reptY = self.plotMoves(event)
            for i in range(reptX):
                canvas.move(object, incrX, 0)
                canvas.update()
                time.sleep(pickDelays[0])
            for i in range(reptY):
                canvas.move(object, 0, incrY)
                canvas.update()                     # update выполняет другие операции
                time.sleep(pickDelays[0])           # sleep до следующего перемещения
            self.moving.remove(object)
            if self.object == object: self.where  = event
    def onSelect(self, event):
        self.where  = event
        self.object = self.canvas.find_closest(event.x, event.y)[0]   # набор
    def onDrag(self, event):
        diffX = event.x - self.where.x              # ok, если объект в движении,
        diffY = event.y - self.where.y              # сбрасывает его с курса
        self.canvas.move(self.object, diffX, diffY)
        self.where = event
    def onOptions(self, event):
        keymap = {
            'w': lambda self: self.changeOption(pickWidths, 'Pen Width'),
            'c': lambda self: self.changeOption(pickFills,  'Color'),
            'u': lambda self: self.changeOption(pickUnits,  'Move Unit'),
            's': lambda self: self.changeOption(pickDelays, 'Move Delay'),
            'x': lambda self: self.changeOption(pickPens,   'Pen Mode'),
            'o': lambda self: self.changeDraw(Canvas.create_oval,      'Oval'),
            'r': lambda self: self.changeDraw(Canvas.create_rectangle, 'Rect'),
            'l': lambda self: self.changeDraw(Canvas.create_line,      'Line'),
            'a': lambda self: self.changeDraw(Canvas.create_arc,       'Arc'),
            'd': MovingPics.deleteObject,
            '1': MovingPics.raiseObject,
            '2': MovingPics.lowerObject,        # если только 1 схема вызова,
            'f': MovingPics.fillObject,         # использовать объекты несвязанных методов,
```

```
        'b': MovingPics.fillBackground,     # иначе лямбде передается self
        'p': MovingPics.addPhotoItem,
        'z': MovingPics.savePostscript,
        '?': MovingPics.help}
    try:
        keymap[event.char](self)
    except KeyError:
        self.setTextInfo('Press ? for help')
def changeDraw(self, method, name):
    self.createMethod = method              # несвязанный метод Canvas
    self.setTextInfo('Draw Object=' + name)
def changeOption(self, list, name):
    list.append(list[0])
    del list[0]
    self.setTextInfo('%s=%s' % (name, list[0]))
def deleteObject(self):
    if self.object != self.textInfo:        # ok, если объект в движении,
        self.canvas.delete(self.object)     # удаляет, но движение продолжается
        self.object = None
def raiseObject(self):
    if self.object:                         # ok, если объект в движении
        self.canvas.tkraise(self.object)    # возбуждает во время движения
def lowerObject(self):
    if self.object:
        self.canvas.lower(self.object)
def fillObject(self):
    if self.object:
        type = self.canvas.type(self.object)
        if type == 'image':
            pass
        elif type == 'text':
            self.canvas.itemconfig(self.object, fill=pickFills[0])
        else:
            self.canvas.itemconfig(self.object,
                           fill=pickFills[0], width=pickWidths[0])
def fillBackground(self):
    self.canvas.config(bg=pickFills[0])
def addPhotoItem(self):
    if not self.where: return
    filetypes=[('Gif files', '.gif'), ('All files', '*')]
    file = askopenfilename(initialdir=PicDir, filetypes=filetypes)
    if file:
        image = PhotoImage(file=file)                       # загрузка образа
        self.images.append(image)                           # сохранить ссылку
        self.object = self.canvas.create_image(             # поместить на холст
                            self.where.x, self.where.y,     # в предыдущую точку
                            image=image, anchor=NW)
def savePostscript(self):
    file = asksaveasfilename()
    if file:
        self.canvas.postscript(file=file)     # сохранить холст в файле
def help(self):
    self.setTextInfo(helpstr)
    #showinfo('PyDraw', helpstr)
def setTextInfo(self, text):
    self.canvas.dchars(self.textInfo, 0, END)
    self.canvas.insert(self.textInfo, 0, text)
    self.canvas.tkraise(self.textInfo)
```

```
    def onQuit(self):
        if self.moving:
            self.setTextInfo("Can't quit while move in progress")
        else:
           self.realquit()  # стандартное wm delete: сообщение об ошибке, если идет перемещение
def traceEvent(label, event, fullTrace=1):
    print label
    if fullTrace:
        for key in dir(event): print key, '=>', getattr(event, key)

if __name__ == '__main__':
    from sys import argv                    # если выполняется этот файл
    if len(argv) == 2: PicDir = argv[1]     # '..' не действует при запуске из другого места
    root = Tk()                             # создать и запустить объект MovingPics
    MovingPics(root)
    root.mainloop()
```

Так же как в примерах `canvasDraw` **из предыдущей главы, можно добавить поддержку одновременного перемещения более чем одного объекта с помощью событий планируемых обратных вызовов** `after` **или потоков. Пример 9.19 показывает подкласс** `MovingPics`**, в котором проведены изменения, необходимые для параллельного перемещения с помощью событий** `after`**. Запустите этот файл непосредственно и увидите разницу; я бы мог попытаться схватить на снимке вид нескольких движущихся объектов, но вряд ли мне это удалось бы.**

Пример 9.19. PP2E\Gui\MovingPics\movingpics_after.py

```
##############################################################################
# PyDraw-after: простая программа рисования и перемещения/анимации объектов
# перемещение объектов осуществляется с помощью планируемых событий .after,
# позволяющих перемещать одновременно несколько объектов, не прибегая к потокам;
# движение выполняется параллельно, но медленнее, чем в версии time.sleep;
# см. также canvasDraw в обзоре: строит и передает сразу весь список incX/incY:
# здесь было бы allmoves = [(incrX, 0)] * reptX) + ([(0, incrY)] * reptY)
##############################################################################

from movingpics import *

class MovingPicsAfter(MovingPics):
    def doMoves(self, delay, objectId, incrX, reptX, incrY, reptY):
        if reptX:
            self.canvas.move(objectId, incrX, 0)
            reptX = reptX - 1
        else:
            self.canvas.move(objectId, 0, incrY)
            reptY = reptY - 1
        if not (reptX or reptY):
            self.moving.remove(objectId)
        else:
            self.canvas.after(delay,
                self.doMoves, delay, objectId, incrX, reptX, incrY, reptY)
    def onMove(self, event):
        traceEvent('onMove', event, 0)
        object = self.object      # переместить текущий объект в точку щелчка
        if object:
            msecs  = int(pickDelays[0] * 1000)
            parms  = 'Delay=%d msec, Units=%d' % (msecs, pickUnits[0])
            self.setTextInfo(parms)
            self.moving.append(object)
```

```
            incrX, reptX, incrY, reptY = self.plotMoves(event)
            self.doMoves(msecs, object, incrX, reptX, incrY, reptY)
            self.where = event

if __name__ == '__main__':
    from sys import argv                        # когда выполняется этот файл
    if len(argv) == 2:
        import movingpics                       # глобальная переменная не этого модуля
        movingpics.PicDir = argv[1]             # и при использовании from* не получится
    root = Tk()                                 # импортировать имена объектов
    MovingPicsAfter(root)
    root.mainloop()
```

Теперь, когда происходит одно или несколько перемещений, можно начать еще одно, щелкнув средней кнопкой по другому объекту и щелкнув правой кнопкой в том месте, куда вы хотите его переместить. Перемещение начинается немедленно, даже если есть другие движущиеся объекты. Запланированные события `after` всех объектов помещаются в одну и ту же очередь цикла событий; Tkinter отправляет их как можно быстрее после срабатывания таймера. Если выполнить этот модуль подкласса непосредственно, то можно заметить, что перемещение не такое плавное и быстрое, как первоначально, но несколько перемещений могут происходить одновременно.

В примере 9.20 показывается, как достичь параллельности с помощью потоков. Такая процедура действует, но как отмечалось в предыдущей главе, обновление GUI в порожденных потоках является, вообще говоря, опасным делом. На моей машине перемещение в этом сценарии с потоками происходит рывками в сравнении с первоначальной версией, что отражает накладные расходы, связанные с переключением интерпретатора (и ЦП) между несколькими потоками.

Пример 9.20. PP2E\Gui\MovingPics\movingpics_threads.py

```
##############################################################################
# перемещение объектов с помощью потоков; должно работать в Windows, если не вызывать
# в потоках canvas.update() (иначе – завершение с фатальными ошибками, нарисованные объекты
# сразу начинают двигаться и т. п.); хотя бы некоторые методы холста в Tkinter должны быть
# безопасными для использования в потоках; менее плавно, чем time.sleep, и опасно
# в целом: лучше всего в потоках обновлять глобальные переменные, не трогая GUI;
##############################################################################

import thread, time, sys, random
from Tkinter import Tk, mainloop
from movingpics import MovingPics, pickUnits, pickDelays

class MovingPicsThreaded(MovingPics):
    def __init__(self, parent=None):
        MovingPics.__init__(self, parent)
        self.mutex = thread.allocate_lock()
        import sys
        #sys.setcheckinterval(0)         # переключение контекста после каждой операции
    def onMove(self, event):            # виртуальной машины не помогает
        object = self.object
        if object and object not in self.moving:
            msecs  = int(pickDelays[0] * 1000)
            parms  = 'Delay=%d msec, Units=%d' % (msecs, pickUnits[0])
            self.setTextInfo(parms)
            #self.mutex.acquire()
            self.moving.append(object)
            #self.mutex.release()
            thread.start_new_thread(self.doMove, (object, event))
```

```
    def doMove(self, object, event):
        canvas = event.widget
        incrX, reptX, incrY, reptY = self.plotMoves(event)
        for i in range(reptX):
            canvas.move(object, incrX, 0)
            # canvas.update()
            time.sleep(pickDelays[0])         # может измениться
        for i in range(reptY):
            canvas.move(object, 0, incrY)
            # canvas.update()                 # update выполняет другие операции
            time.sleep(pickDelays[0])         # спать до следующего перемещения
        #self.mutex.acquire()
        self.moving.remove(object)
        if self.object == object: self.where  = event
        #self.mutex.release()

if __name__ == '__main__':
    root = Tk()
    MovingPicsThreaded(root)
    mainloop()
```

PyClock: графический элемент аналоговых/цифровых часов

Изучая новый интерфейс компьютера, я всегда вначале отыскиваю часы. Я столько времени неотрывно нахожусь за компьютером, что мне совершенно невозможно следить за временем, если не видеть его прямо перед собой на экране (и даже тогда это проблематично). Следующая программа, PyClock, реализует такой графический элемент часов на Python. Она не очень отличается от тех часов, которые вы привыкли видеть в системах X Windows. Но так как ее код написан на Python, то она легко перенастраиваема и переносима между Windows, X Windows и Macintosh, как и весь код этой главы. В дополнение к развитым технологиям GUI этот пример демонстрирует использование модулей Python `math` и `time`.

Краткий урок геометрии

Прежде чем показывать вам PyClock, немного предыстории и признаний. Ну-ка, ответьте: как поставить точки на окружности? Эта задача, а также форматы времени и события оказываются основными при создании графических элементов часов. Чтобы нарисовать аналоговый циферблат часов на элементе холста, необходимо уметь нарисовать круг – сам циферблат состоит из точек окружности, а секундная, минутная и часовая стрелки представляют собой линии, проведенные из центра в точки окружности. Цифровые часы нарисовать проще, но смотреть на них неинтересно.

Теперь признание: начав писать PyClock, я не знал ответа на первый вопрос предыдущего параграфа. Я совершенно забыл математику построения точек окружности (как и большинство профессиональных программистов, у которых я этим интересовался). Бывает. Такие знания, не будучи востребованными в течение нескольких десятилетий, могут попасть в уборку мусора. В конце концов мне удалось смахнуть пыль с нескольких нейронов, длины которых оказалось достаточно, чтобы запрограммировать необходимые для построения действия, но блеснуть умом мне не удалось.

Если вы в таком же положении, то я покажу вам один способ простой записи на Python формул построения точек, но для глубоких занятий геометрией места здесь нет. Прежде чем взяться за более сложную задачу реализации часов, я написал сценарий

plotterGui, приведенный в примере 9.21, чтобы сосредоточиться только на логике построения круга.

Логика круга заключена в функции point – она строит координаты (X,Y) точки окружности по ее относительному номеру, общему количеству точек, помещаемых на окружности, и радиусу окружности (расстоянию между центром окружности и ее точками). Сначала вычисляется угол точки от вершины путем деления 360 на количество рисуемых точек и умножения на номер точки. Напомню, что весь круг составляет 360 градусов (например, если на окружности рисуется 4 точки, то каждая отстоит от предыдущей на 90 градусов, или 360/4). Стандартный модуль Python math предоставляет все необходимые для дальнейшего константы и функции – *pi*, *sine* и *cosine*. В действительности математика тут не слишком непонятная, если достаточно внимательно ее рассмотреть (возможно, вместе со старым учебником геометрии). На прилагаемом к книге CD можно найти альтернативные способы кодирования обработки чисел.[1]

Даже если вас не интересует математика, посмотрите в сценарии на функцию circle. Получив координаты (X,Y) точки окружности, возвращенные point, она чертит линию из центра окружности в точку и маленький прямоугольник вокруг самой точки, что несколько напоминает стрелки и отметки аналоговых часов. На холсте используются теги, чтобы удалять нарисованные объекты перед каждым построением.

Пример 9.21. PP2E\Gui\Clock\plotterGui.py

```
# рисование кругов (как я делал это в старших классах)

import math, sys
from Tkinter import *

def point(tick, range, radius):
    angle = tick * (360.0 / range)
    radiansPerDegree = math.pi / 180
    pointX = int( round( radius * math.sin(angle * radiansPerDegree) ))
    pointY = int( round( radius * math.cos(angle * radiansPerDegree) ))
    return (pointX, pointY)

def circle(points, radius, centerX, centerY, slow=0):
    canvas.delete('lines')
    canvas.delete('points')
    for i in range(points):
        x, y = point(i+1, points, radius-4)
        scaledX, scaledY = (x + centerX), (centerY - y)
        canvas.create_line(centerX, centerY, scaledX, scaledY, tag='lines')
        canvas.create_rectangle(scaledX-2, scaledY-2,
                                scaledX+2, scaledY+2,
                                           fill='red', tag='points')
        if slow: canvas.update()

def plotter():
    circle(scaleVar.get(), (Width / 2), originX, originY, checkVar.get())
```

[1] Если вы достаточно много занимаетесь обработкой чисел и разобрались в этом параграфе, вас, возможно, заинтересует NumPy – расширение Python для численного программирования. Там есть такие объекты, как векторы и сложные математические операции, превращающие Python в инструмент научного программирования. Это расширение эффективно используется многими организациями, в том числе Ливерморской национальной лабораторией. NumPy нужно получить и установить отдельно; смотрите ссылки на сайте Python. Кроме того, в Python есть встроенный тип комплексных чисел для инженерных расчетов; подробности смотрите в руководстве по библиотеке.

```
def makewidgets():
    global canvas, scaleVar, checkVar
    canvas = Canvas(width=Width, height=Width)
    canvas.pack(side=TOP)
    scaleVar = IntVar()
    checkVar = IntVar()
    scale = Scale(label='Points on circle', variable=scaleVar, from_=1, to=360)
    scale.pack(side=LEFT)
    Checkbutton(text='Slow mode', variable=checkVar).pack(side=LEFT)
    Button(text='Plot', command=plotter).pack(side=LEFT, padx=50)

if __name__ == '__main__':
    Width = 500                                          # ширина, высота по умолчанию
    if len(sys.argv) == 2: Width = int(sys.argv[1])  # ширина в командной строке?
    originX = originY = Width / 2                        # то же, что радиус
    makewidgets()                                        # в корне Tk по умолчанию
    mainloop()
```

По умолчанию ширина круга составит 500 пикселов, если не задать ширину в командной строке. Получив число точек на окружности, этот сценарий размечает окружность против часовой стрелки при каждом нажатии Plot, вычерчивая прямые из центра к маленьким прямоугольникам в точках на окружности. Переместите ползунок, чтобы нарисовать другое число точек, и щелкните по флажку, чтобы рисование происходило достаточно медленно и можно было заметить очередность вычерчивания линий и точек (при этом сценарий выполняет `update` для обновления экрана после каждого вычерчивания линии). Рис. 9.19 показывает результат нанесения 120 отметок при установке ширины круга в командной строке, равной 400; если потребовать 60 или 12 точек на окружности, сходство с часовым циферблатом становится более заметным.

Дополнительную помощь могут оказать содержащиеся на CD текстовые версии этого сценария графики, которые выводят координаты точек окружности в поток `stdout`, а

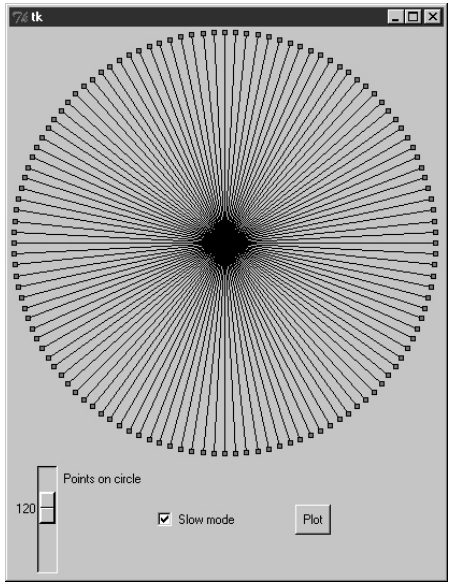


Рис. 9.19. plotterGui в действии

не отображают в GUI. Смотрите сценарии plotterText в каталоге часов. Вот что они показывают для 4 и 12 точек на окружности шириной 400 точек; формат вывода:

```
pointnumber : angle = (Xcoordinate, Ycoordinate)
```

Предполагается, что центр круга имеет координаты (0,0):

```
----------
1 : 90.0 = (200, 0)
2 : 180.0 = (0, -200)
3 : 270.0 = (-200, 0)
4 : 360.0 = (0, 200)
----------
1 : 30.0 = (100, 173)
2 : 60.0 = (173, 100)
3 : 90.0 = (200, 0)
4 : 120.0 = (173, -100)
5 : 150.0 = (100, -173)
6 : 180.0 = (0, -200)
7 : 210.0 = (-100, -173)
8 : 240.0 = (-173, -100)
9 : 270.0 = (-200, 0)
10 : 300.0 = (-173, 100)
11 : 330.0 = (-100, 173)
12 : 360.0 = (0, 200)
----------
```

Чтобы понять, как эти точки отображаются на холст, запомните сначала, что ширина и высота круга всегда одинаковы – радиус × 2. Поскольку координаты холста Tkinter (X,Y) начинаются с (0,0) в левом верхнем углу, построитель GUI должен сместить центр окружности в точку с координатами (ширина/2, ширина/2) – начальную точку, из которой вычерчиваются прямые. Например, в круге 400 на 400 центр холста будет в (200,200). Прямая в точку с углом 90 градусов с правой стороны окружности проходит из (200,200) в (400,200) – результат добавления координат точки (200,0), полученной для данного радиуса и угла. Линия вниз под углом 180 градусов проходит из (200,200) в (200,400) после учета выведенной точки (0,–200).

Этот алгоритм построения точек, применяемый в plotterGui, а также несколько констант масштабирования лежат в сердце аналогового изображения PyClock. Если все же вам кажется, что это слишком сложно, предлагаю сначала сосредоточиться на реализации *цифрового* экрана сценария PyClock. Аналоговые геометрические построения в действительности лишь расширяют механизм отсчета времени, использующийся в обоих режимах отображения. В действительности сами часы имеют структуру общего объекта Frame, одинаковым образом посылающего *встроенным* объектам цифрового и аналогового отображения события изменения времени и размеров. Аналоговый дисплей – это прикрепленный Canvas, умеющий рисовать окружности, а цифровой объект – просто прикрепленный Frame, метки которого показывают время.

Выполнение PyClock

За исключением части, касающейся построения окружностей, код PyClock простой. Он рисует циферблат для показа текущего времени и с помощью методов after будит себя 10 раз в секунду, проверяя, не перевалило ли системное время на следующую секунду. Если да, то перерисовываются секундная, минутная и часовая стрелки, чтобы показать новое время (либо изменяется текст меток цифрового дисплея). На языке создания GUI это означает, что аналоговое изображение выводится на холсте, перерисовывается при изменении размеров окна и при запросе изменяется на цифровой формат.

В PyClock также применяется стандартный модуль Python `time`, чтобы получать и преобразовывать системные данные времени, необходимые для часов. Если описать кратко, то метод `onTimer` получает системное время через `time.time`, встроенное средство, возвращающее число с плавающей точкой, выражающее количество секунд, прошедших с начала «эпохи» – точки начала отсчета времени на вашем компьютере. Затем это время преобразуется в набор, состоящий из значений часов, минут и секунд, с помощью вызова `time.localtime`; дополнительные подробности, относящиеся к работе с временем, можно найти в самом сценарии и руководстве по библиотеке Python.

Проверка системного времени 10 раз в секунду может показаться чрезмерной, но она гарантирует перемещение секундной стрелки вовремя и без рывков и скачков (события `after` синхронизируются не очень точно) и не влечет существенного отвлечения мощности ЦП на тех машинах, которыми я пользуюсь.[1] В Linux и в Windows PyClock незначительно расходует ресурсы процессора – в основном при обновлении экрана в аналоговом режиме, а не в событиях `after`. Чтобы минимизировать обновления экрана, PyClock перерисовывает часовые стрелки только при скачках секунд; отметки на циферблате перерисовываются только при начальном запуске и изменении размеров окна. На рис. 9.20 показан начальный формат вывода PyClock по умолчанию, когда *clock.py* запускается непосредственно.

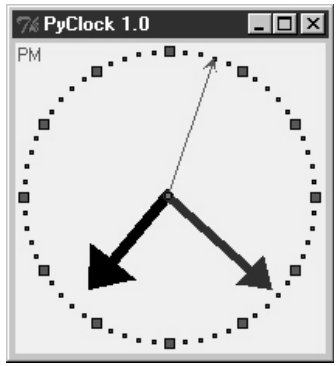


Рис. 9.20. Аналоговый экран PyClock по умолчанию

Линии, представляющие стрелки часов, действительно имеют форму стрелок, что определяется параметрами объекта линии `arrow` и `arrowshape`. Параметр `arrow` может принимать значения «first», «last», «none» или «both»; параметр `arrowshape` задается как набор чисел, задающих длину стрелки, касающейся линии, ее общую длину и ее ширину.

Как и PyView, PyClock динамически удаляет и перерисовывает части изображения по требованию (то есть в ответ на связанные события) с помощью методов `pack_forget` и `pack`. Щелчок по часам левой кнопкой мыши изменяет формат вывода на цифровой путем удаления графических элементов аналогового режима и вывода цифрового интерфейса; в результате получается более простой интерфейс, показанный на рис. 9.21.

Такая цифровая форма вывода полезна, если вы хотите сберечь драгоценное место на экране и уменьшить использование ЦП (расходы на обновление экрана очень малы).

---

[1] Что касается производительности, то я запускал по нескольку часов на всех проверяемых машинах – от Pentium III 650 МГц до «старого» Pentium I 200 МГц – и не обнаружил ухудшения работы каких-либо из выполнявшихся часов. Например, сценарий PyDemos запускает шесть часов, выполняющихся в одном процессе, и во всех них обновление времени происходит плавно. Возможно, так же обстоит дело и на более старых машинах, но мои настолько покрылись пылью, что полезных измерений на них уже не произвести.

Щелчок по часам левой кнопкой снова переводит их в аналоговый режим отображения. При запуске сценария строятся оба отображения – аналоговое и цифровое, но в каждый данный момент упаковано только одно из них.

Рис. 9.21. PyClock стал цифровым

Щелчок по часам правой кнопкой мыши в любом из режимов отображения вызывает появление или исчезновение прикрепленной метки, показывающей текущую дату в простом текстовом формате. На рис. 9.22 показан PyClock, выполняющийся в аналоговом режиме, с меткой даты и размещенным по центру фотографическим изображением.

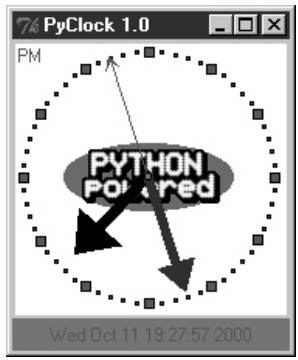


Рис. 9.22. Расширенное изображение PyClock с графикой

Изображение в центре добавлено путем передачи конструктору объекта PyClock объекта конфигурации с надлежащими установками. В действительности почти все особенности изображения можно настроить с помощью атрибутов объектов конфигурации PyClock – цвет стрелок, цвет меток, центральное изображение и начальный размер.

Так как в аналоговом режиме PyClock фигуры вручную рисуются на холсте, необходимо самостоятельно обрабатывать события *изменения размеров* окна: когда окно уменьшается или увеличивается, нужно перерисовывать циферблат часов в соответствии с новыми размерами окна. Для перехвата изменений размеров окна сценарий регистрирует событие `<Configure>` с помощью `bind`; удивительно, но это не событие менеджера верхнего окна, как для кнопки close. Если расширить PyClock, то циферблат увеличится вместе с окном, – попробуйте расширить, сжать и развернуть окно часов во весь экран на своем компьютере. Так как циферблат строится в квадратной системе координат, PyClock всегда расширяется в равном отношении по вертикали и горизонтали; если просто сделать окно только шире или выше, циферблат не изменится.

Наконец, подобно PyEdit, можно запускать PyClock автономно или прикрепленным и встроенным в другой GUI, в котором требуется вывести текущее время. Для облегчения запуска сконфигурированных некоторым образом часов существует вспомогательный модуль `clockStyles`, предоставляющий ряд объектов конфигурации, которые можно импортировать, расширять, создавая подклассы, и передавать в конструктор часов. На рис. 9.23 показано несколько заранее подготовленных стилей и размеров часов в действии, ведущих синхронный отсчет времени.

Во всех этих часах 10 раз в секунду проверяется изменение системного времени с использованием событий `after`. При выполнении в качестве окон верхнего уровня в од-

ном и том же процессе все они получают событие таймера из одного и того же цикла событий. При запуске в качестве независимых программ у каждой из них имеется собственный цикл событий. В том и другом случае их секундные стрелки раз в секунду дружно перемещаются.

Рис. 9.23. Несколько готовых стилей часов (фото Гвидо перепечатано с разрешения «Dr. Dobb's Journal»)

Исходный код PyClock

Весь исходный код PyClock располагается в одном файле, за исключением объектов предварительно закодированных стилей конфигурации. Если посмотреть в конец кода примера 9.22, то можно заметить, что объект часов можно создать либо передав объект конфигурации, либо задав параметры конфигурации аргументами командной строки (и тогда сценарий просто сам построит объект конфигурации). Вообще говоря, можно непосредственно выполнить этот файл для запуска часов, импортировать его и создать объекты с использованием объектов конфигурации, чтобы часы выглядели более индивидуально, или импортировать и прикрепить его объекты к другому GUI. Например, PyGadgets выполняет этот файл с параметрами командной строки для индивидуальной настройки внешнего вида.

Пример 9.22. PP2E\Gui\Clock\clock.py

```
##############################################################################
# PyClock: GUI часов с аналоговым и числовым режимами отображения,
# всплывающей меткой даты, графикой циферблата, изменением размеров и т. д.
# может выполняться самостоятельно или встроенным (прикрепленным) в другой GUI.
##############################################################################
```

```
from Tkinter import *
import math, time, string

##############################################################################
# Дополнительные конфигурирующие классы
##############################################################################

class ClockConfig:
    # defaults--override in instance or subclass
    size = 200                                      # ширина=высота
    bg, fg = 'beige', 'brown'                       # цвета циферблата, меток
    hh, mh, sh, cog = 'black', 'navy', 'blue', 'red'  # стрелок, центра
    picture = None                                  # файл картинки для циферблата

class PhotoClockConfig(ClockConfig):
    # пример конфигурации
    size    = 320
    picture = '../gifs/ora-pp.gif'
    bg, hh, mh = 'white', 'blue', 'orange'

##############################################################################
# Объект цифрового отображения
##############################################################################

class DigitalDisplay(Frame):
    def __init__(self, parent, cfg):
        Frame.__init__(self, parent)
        self.hour = Label(self)
        self.mins = Label(self)
        self.secs = Label(self)
        self.ampm = Label(self)
        for label in self.hour, self.mins, self.secs, self.ampm:
            label.config(bd=4, relief=SUNKEN, bg=cfg.bg, fg=cfg.fg)
            label.pack(side=LEFT)

    def onUpdate(self, hour, mins, secs, ampm, cfg):
        mins = string.zfill(str(mins), 2)
        self.hour.config(text=str(hour), width=4)
        self.mins.config(text=str(mins), width=4)
        self.secs.config(text=str(secs), width=4)
        self.ampm.config(text=str(ampm), width=4)

    def onResize(self, newWidth, newHeight, cfg):
        pass  # перерисовывать нечего


##############################################################################
# Объект аналогового отображения
##############################################################################

class AnalogDisplay(Canvas):
    def __init__(self, parent, cfg):
        Canvas.__init__(self, parent,
                        width=cfg.size, height=cfg.size, bg=cfg.bg)
        self.drawClockface(cfg)
        self.hourHand = self.minsHand = self.secsHand = self.cog = None

    def drawClockface(self, cfg):                       # при запуске и изменении размеров
        if cfg.picture:                                 # рисовать овал, картинку
            try:
                self.image = PhotoImage(file=cfg.picture)          # фон
```

```
            except:
                self.image = BitmapImage(file=cfg.picture)         # сохранить ссылку
            imgx = (cfg.size - self.image.width())  / 2            # центрировать
            imgy = (cfg.size - self.image.height()) / 2
            self.create_image(imgx+1, imgy+1,  anchor=NW, image=self.image)
        originX =  originY = radius = cfg.size/2
        for i in range(60):
            x, y = self.point(i, 60, radius-6, originX, originY)
            self.create_rectangle(x-1, y-1, x+1, y+1, fill=cfg.fg)  # минуты
        for i in range(12):
            x, y = self.point(i, 12, radius-6, originX, originY)
            self.create_rectangle(x-3, y-3, x+3, y+3, fill=cfg.fg)  # часы
        self.ampm = self.create_text(3, 3, anchor=NW, fill=cfg.fg)

    def point(self, tick, units, radius, originX, originY):
        angle = tick * (360.0 / units)
        radiansPerDegree = math.pi / 180
        pointX = int( round( radius * math.sin(angle * radiansPerDegree) ))
        pointY = int( round( radius * math.cos(angle * radiansPerDegree) ))
        return (pointX + originX+1), (originY+1 - pointY)

    def onUpdate(self, hour, mins, secs, ampm, cfg):        # по обратному вызову таймера
        if self.cog:                                        # перерисовать стрелки, центр
            self.delete(self.cog)
            self.delete(self.hourHand)
            self.delete(self.minsHand)
            self.delete(self.secsHand)
        originX = originY = radius = cfg.size/2
        hour = hour + (mins / 60.0)
        hx, hy = self.point(hour, 12, (radius * .80), originX, originY)
        mx, my = self.point(mins, 60, (radius * .90), originX, originY)
        sx, sy = self.point(secs, 60, (radius * .95), originX, originY)
        self.hourHand = self.create_line(originX, originY, hx, hy,
                            width=(cfg.size * .04),
                            arrow='last', arrowshape=(25,25,15), fill=cfg.hh)
        self.minsHand = self.create_line(originX, originY, mx, my,
                            width=(cfg.size * .03),
                            arrow='last', arrowshape=(20,20,10), fill=cfg.mh)
        self.secsHand = self.create_line(originX, originY, sx, sy,
                            width=1,
                            arrow='last', arrowshape=(5,10,5), fill=cfg.sh)
        cogsz = cfg.size * .01
        self.cog = self.create_oval(originX-cogsz, originY+cogsz,
                                    originX+cogsz, originY-cogsz, fill=cfg.cog)
        self.dchars(self.ampm, 0, END)
        self.insert(self.ampm, END, ampm)

    def onResize(self, newWidth, newHeight, cfg):
        newSize = min(newWidth, newHeight)
        #print 'analog onResize', cfg.size+4, newSize
        if newSize != cfg.size+4:
            cfg.size = newSize-4
            self.delete('all')
            self.drawClockface(cfg)  # onUpdate called next

################################################################################
# Составной объект часов
################################################################################
```

```
ChecksPerSec = 10  # second change timer

class Clock(Frame):
    def __init__(self, config=ClockConfig, parent=None):
        Frame.__init__(self, parent)
        self.cfg = config
        self.makeWidgets(parent)                    # дочерние пакуются,
        self.labelOn = 0                            # но клиенты могут делать pack или grid
        self.display = self.digitalDisplay
        self.lastSec = -1
        self.onSwitchMode(None)
        self.onTimer()

    def makeWidgets(self, parent):
        self.digitalDisplay = DigitalDisplay(self, self.cfg)
        self.analogDisplay  = AnalogDisplay(self,  self.cfg)
        self.dateLabel      = Label(self, bd=3, bg='red', fg='blue')
        parent.bind('<ButtonPress-1>', self.onSwitchMode)
        parent.bind('<ButtonPress-3>', self.onToggleLabel)
        parent.bind('<Configure>',     self.onResize)

    def onSwitchMode(self, event):
        self.display.pack_forget()
        if self.display == self.analogDisplay:
            self.display = self.digitalDisplay
        else:
            self.display = self.analogDisplay
        self.display.pack(side=TOP, expand=YES, fill=BOTH)

    def onToggleLabel(self, event):
        self.labelOn = self.labelOn + 1
        if self.labelOn % 2:
            self.dateLabel.pack(side=BOTTOM, fill=X)
        else:
            self.dateLabel.pack_forget()
        self.update()

    def onResize(self, event):
        if event.widget == self.display:
            self.display.onResize(event.width, event.height, self.cfg)

    def onTimer(self):
        secsSinceEpoch = time.time()
        timeTuple      = time.localtime(secsSinceEpoch)
        hour, min, sec = timeTuple[3:6]
        if sec != self.lastSec:
            self.lastSec = sec
            ampm = ((hour >= 12) and 'PM') or 'AM'              # 0...23
            hour = (hour % 12) or 12                            # 12..11
            self.display.onUpdate(hour, min, sec, ampm, self.cfg)
            self.dateLabel.config(text=time.ctime(secsSinceEpoch))
        self.after(1000 / ChecksPerSec, self.onTimer)   # выполнять N раз в секунду

################################################################################
# Автономные часы
################################################################################

class ClockWindow(Clock):
    def __init__(self, config=ClockConfig, parent=None, name=''):
        Clock.__init__(self, config, parent)
```

```
        self.pack(expand=YES, fill=BOTH)
        title = 'PyClock 1.0'
        if name: title = title + ' - ' + name
        self.master.title(title)               # master=родитель или по умолчанию
        self.master.protocol('WM_DELETE_WINDOW', self.quit)

##############################################################################
# Выполнение программы
##############################################################################

if __name__ == '__main__':
    def getOptions(config, argv):
        for attr in dir(ClockConfig):             # заполнить объект конфигурации по умолчанию
            try:                                  # из аргументов командной строки "-attr val"
                ix = argv.index('-' + attr)
            except:
                continue
            else:
                if ix in range(1, len(argv)-1):
                    if type(getattr(ClockConfig, attr)) == type(0):
                        setattr(config, attr, int(argv[ix+1]))
                    else:
                        setattr(config, attr, argv[ix+1])
    import sys
    config = ClockConfig()
    #config = PhotoClockConfig()
    if len(sys.argv) >= 2:
        getOptions(config, sys.argv)         # clock.py -size n -bg 'blue'...
    myclock = ClockWindow(config, Tk())      # при автономном выполнении родителем
                                             # служит корень Tk
    myclock.mainloop()
```

И наконец, в примере 9.23 показан модуль, фактически выполняемый из запускающего сценария PyDemos, – в нем предопределен ряд стилей часов и производится запуск одновременно шести часов, прикрепляемых к новым окнам верхнего уровня для создания демонстрационного эффекта (хотя на практике обычно даже мне достаточно иметь на экране одни часы).[1]

Пример 9.23. PP2E\Gui\Clock\clockStyles.py

```
from clock import *
from Tkinter import mainloop

gifdir = '../gifs/'
if __name__ == '__main__':
    from sys import argv
    if len(argv) > 1:
        gifdir = argv[1] + '/'

class PPClockBig(PhotoClockConfig):
    picture, bg, fg = gifdir + 'ora-pp.gif', 'navy', 'green'

class PPClockSmall(ClockConfig):
    size    = 175
    picture = gifdir + 'ora-pp.gif'
```

[1] **Учтите, что указанные в этом сценарии графические файлы могут отсутствовать на CD из соображений соблюдения авторских прав. Можете в отместку вспомнить здесь свой любимый анекдот про юриста.**

```
    bg, fg, hh, mh = 'white', 'red', 'blue', 'orange'

class GilliganClock(ClockConfig):
    size    = 550
    picture = gifdir + 'gilligan.gif'
    bg, fg, hh, mh = 'black', 'white', 'green', 'yellow'

class GuidoClock(GilliganClock):
    size = 400
    picture = gifdir + 'guido_ddj.gif'
    bg = 'navy'

class GuidoClockSmall(GuidoClock):
    size, fg = 278, 'black'

class OusterhoutClock(ClockConfig):
    size, picture = 200, gifdir + 'ousterhout-new.gif'
    bg, fg, hh    = 'black', 'gold', 'brown'

class GreyClock(ClockConfig):
    bg, fg, hh, mh, sh = 'grey', 'black', 'black', 'black', 'white'

class PinkClock(ClockConfig):
    bg, fg, hh, mh, sh = 'pink', 'yellow', 'purple', 'orange', 'yellow'

class PythonPoweredClock(ClockConfig):
    bg, size, picture = 'white', 175, gifdir + 'pythonPowered.gif'

if __name__ == '__main__':
    for configClass in [
        ClockConfig,
        PPClockBig,
        #PPClockSmall,
        GuidoClockSmall,
        #GilliganClock,
        OusterhoutClock,
        #GreyClock,
        PinkClock,
        PythonPoweredClock
    ]:
        ClockWindow(configClass, Toplevel(), configClass.__name__)
    Button(text='Quit Clocks', command='exit').pack()
    mainloop()
```

PyToe: графический элемент игры в крестики-нолики

И наконец, в завершение главы немного развлечений. В нашем последнем примере, *PyToe*, с помощью Python реализована программа игры в крестики-нолики («tic-tac-toe») с искусственным интеллектом. Большинству читателей, вероятно, знакома эта игра, поэтому я не стану останавливаться на ее подробностях. Если вкратце, то игроки поочередно ставят свои метки в клетках игровой доски, пытаясь занять целиком строку, колонку или диагональ. Победителем является тот, кому удалось первым это сделать.

В PyToe позиции на доске помечаются щелчком мыши, а одним из игроков является программа Python. Сама игровая доска выводится с помощью простого GUI Tkinter; по умолчанию PyToe строит доску размером 3 на 3 (стандартная настройка игры), но можно настроиться на игру произвольного размера N на N.

Когда приходит очередь компьютера сделать ход, с помощью алгоритмов искусственного интеллекта (AI) оцениваются возможные ходы и ведется поиск в дереве этих ходов и возможных ответов на них. Это довольно простая задача для игровых программ, а эвристики, применяемые для выбора ходов, несовершенны. Все же PyToe обычно достаточно сообразителен, чтобы найти выигрыш на несколько ходов раньше, чем пользователь.

Выполнение PyToe

GUI PyToe реализован в виде фрейма с упакованными метками и привязкой щелчков мыши к этим меткам для перехвата ходов пользователя. Текст метки устанавливается равным метке игрока после каждого хода компьютера или пользователя. Класс `GuiMaker`, который мы кодировали ранее в этой главе, тоже использован здесь для создания простой панели меню в верхней части окна (но панель инструментов внизу не выводится, так как PyToe оставляет ее дескриптор пустым). По умолчанию пользователь ставит метки «X», а PyToe ставит «O». На рис. 9.24 показано, как PyToe готов победить меня одним из двух способов.

Рис. 9.25 показывает всплывающий диалог подсказки PyToe, в котором приведены параметры конфигурации командной строки. Есть возможность задать цвет и размер для меток игровой доски, игрока, делающего первый ход, метку пользователя («X» или «O»), размер доски (переопределяющий 3 на 3 по умолчанию) и стратегию выбора хода для компьютера (например, «Minimax» выполняет поиск выигрышей и поражений в дереве ходов, а «Expert1» и «Expert2» используют статические эвристические функции оценки).

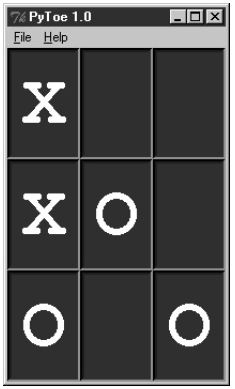


Рис. 9.24. PyToe обдумывает путь к победе

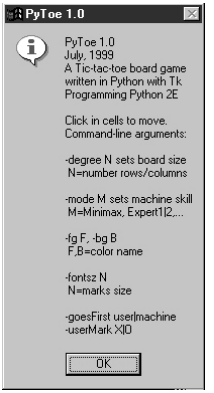


Рис. 9.25. Окно подсказки PyToe с параметрами командной строки

Используемая в PyToe технология ИИ интенсивно использует ЦП, и одни стратегии выбора хода компьютером требуют большего времени, чем другие, но скорость зависит в основном от скорости компьютера. Задержка, связанная с выбором хода, составляет на моей машине (650 МГц) доли секунды при игре 3 на 3 для любого задания стратегии выбора хода «-mode».

На рис. 9.26 показана альтернативная конфигурация PyToe сразу после того, как программа выиграла у меня. Несмотря отобранные для этой книги картинки, при некоторых параметрах выбора хода мне все же удается иногда выигрывать. При большей доске и более сложной игре алгоритм выбора хода PyToe становится еще более эффективным.

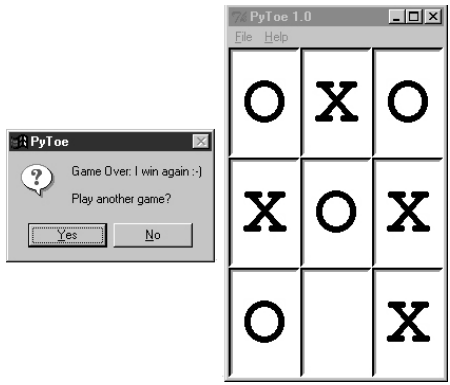


Рис. 9.26. Вариант размещения

Исходный код PyToe (на CD)

PyToe является крупной системой, предполагающей наличие некоторой подготовки в ИИ, а в отношении GUI, в сущности, не демонстрирует ничего нового. Отчасти по этой причине, но в основном из-за того, что я уже исчерпал предел страниц, отведенных на эту главу, исходный код не будет здесь приводиться, а обратиться за ним следует к прилагаемому CD. За деталями реализации PyToe обратитесь, пожалуйста, к следующим двум файлам из дистрибутива примеров:

- *PP2E\Ai\TicTacToe\tictactoe.py*, сценарий оболочки верхнего уровня
- *PP2E\Ai\TicTacToe\tictactoe_lists.py*, суть реализации

Могу вам посоветовать обратить внимание на то, что наибольшую трудность составляет структура данных используемой для представления состояния игровой доски. Если вы разберетесь, каким образом моделируется доска, то остальная часть кода станет вполне понятна.

Например, в варианте, основанном на списках, для представления состояния доски используется список списков, а также простой словарь из графических элементов полей ввода для GUI, индексируемый координатами игровой доски. Очистка доски после игры заключается просто в очистке исходных структур данных, как показано в следующем отрывке кода из указанных выше примеров:

```
def clearBoard(self):
    for row, col in self.label.keys():
        self.board[row][col] = Empty
        self.label[(row, col)].config(text=' ')
```

Аналогично выбор хода, по крайней мере, в случайном режиме, заключается в том, чтобы найти пустую ячейку в массиве, представляющем доску, и записать метку машины там и в GUI (degree означает размер доски):

```
def machineMove(self):
    row, col = self.pickMove()
    self.board[row][col] = self.machineMark
    self.label[(row, col)].config(text=self.machineMark)

def pickMove(self):
    empties = []
    for row in self.degree:
```

```
            for col in self.degree:
                if self.board[row][col] == Empty:
                    empties.append((row, col))
        return random.choice(empties)
```

Наконец, проверка состояния завершения игры сводится к просмотру строк, колонок и диагоналей по следующей схеме:

```
    def checkDraw(self, board=None):
        board = board or self.board
        for row in board:
            if Empty in row:
                return 0
        return 1 # не пусто: ничья или выигрыш

    def checkWin(self, mark, board=None):
        board = board or self.board
        for row in board:
            if row.count(mark) == self.degree:      # проверка по горизонтали
                return 1
        for col in range(self.degree):
            for row in board:                       # проверка по вертикали
                if row[col] != mark:
                    break
            else:
                return 1
        for row in range(self.degree):              # проверить диаг1
            col = row                               # строка == колонка
            if board[row][col] != mark: break
        else:
            return 1
        for row in range(self.degree):              # проверить диаг2
            col = (self.degree-1) - row             # row+col = degree-1
            if board[row][col] != mark: break
        else:
            return 1

    def checkFinish(self):
        if self.checkWin(self.userMark):
            outcome = "You've won!"
        elif self.checkWin(self.machineMark):
            outcome = 'I win again :-)'
        elif self.checkDraw():
            outcome = 'Looks like a draw'
```

Другой код выбора хода в основном просто проводит другие виды анализа структуры данных игровой доски или генерирует новые состояния доски для поиска в дереве ходов и ответных ходов.

В том же каталоге находятся родственные файлы, реализующие альтернативные схемы поиска и оценки ходов, различные представления доски и т. д. За дополнительными сведениями об оценке игры и поиске в целом обратитесь к учебникам по ИИ. Это интересный материал, но слишком сложный, чтобы можно было достаточным образом осветить его в данной книге.

Что дальше

На этом завершается часть данной книги, посвященная GUI, но рассказ о GUI здесь не кончается. Если хотите лучше изучить GUI, посмотрите примеры Tkinter, которые будут встречаться дальше в книге и описаны в начале этой главы. PyMail, PyCalc, PyForm и PyTree – все они представляют собой дополнительные конкретные примеры GUI. В следующей части книги мы узнаем также, как создавать интерфейсы пользователя, выполняемые в веб-броузерах, – совершенно другая идея, но еще один вариант конструкции простых интерфейсов.

Помните также, что даже если вы не найдете в этой книге примера GUI, который выглядит очень похожим на тот, который вам нужно запрограммировать, то все конструктивные элементы вам уже знакомы. Создание более крупного GUI для вашего приложения в действительности заключается в иерархическом расположении составляющих его графических элементов, представленных в этой части книги.

Например, сложный экран может быть составлен в виде совокупности радиокнопок, окон списков, ползунков, текстовых полей, меню и т. д., располагаемых во фреймах или на сетке для получения требуемого внешнего вида. Сложный графический интерфейс может быть дополнен всплывающими окнами верхнего уровня, а также независимо выполняемыми программами GUI, связь с которыми поддерживается через такие механизмы IPC, как каналы, сигналы и сокеты.

Кроме того, более крупные компоненты GUI могут быть реализованы в виде классов Python, прикрепляемых или расширяемых всюду, где требуется аналогичный инструмент интерфейса (важным примером этого служит PyEdit). Немного творчества, и с помощью набора графических элементов Tkinter и Python можно создавать практически неограниченное число структур.

Помимо данной книги посмотрите также раздел, посвященный документации и книгам, на сайте Python *http://www.python.org*. Можно было бы поместить сюда тексты, относящиеся к Tkinter, но, надо полагать, предлагаемое в этом разделе расширится за то время, пока актуальна данная книга. Наконец, если вас заразит Tkinter, еще раз хочу порекомендовать загрузить пакеты, о которых было сказано в главе 6, особенно PMW и PIL, и поэкспериментировать с ними. Оба они наращивают мощь арсенала Tkinter, позволяя создавать более сложные GUI при минимуме кодирования.

III

Создание сценариев для Интернета

В этой части книги рассматриваются роль Python как языка программирования приложений для Интернета и средства его библиотеки для поддержки этой роли. Попутно привлекаются к использованию инструменты GUI, представленные ранее в книге. Поскольку это популярная область применения Python, главы данной части охватывают все направления:

- Глава 10 «Сетевые сценарии». Здесь происходит знакомство с Интернетом, представлены средства Python сетевых коммуникаций низкого уровня, такие как сокеты и вызовы select, а также иллюстрируется стандартная техника программирования архитектуры клиент–сервер на Python.
- Глава 11 «Сценарии на стороне клиента». В этой главе показано, как с помощью Python сценарии могут обращаться к стандартным сетевым протоколам клиента, таким как FTP, e-mail[1], HTTP и другие.
- Глава 12 «Сценарии на стороне сервера». Эта глава освещает основы CGI-сценариев Python, выполняемых на сервере, – программ, используемых при создании интерактивных веб-сайтов.
- Глава 13 «Крупные примеры веб-сайтов, часть 1». Эта глава демонстрирует приемы реализации веб-сайтов с помощью Python, такие как обеспечение защиты, на примере основанной на веб почтовой системы.
- Глава 14 «Крупные примеры веб-сайтов, часть 2». В этой главе представлены дополнительные приемы создания веб-сайтов, в том числе постоянное хранение данных на сервере, на примере системы сообщения о найденных ошибках.
- Глава 15 «Более сложные темы Интернета». В этой главе дается обзор систем и средств Python для создания интернет-приложений, таких как JPython, Active Scripting, Zope и HTMLgen.

---

[1] На самом деле «протокол e-mail» не существует. Здесь автор имеет в виду набор протоколов, используемых для передачи электронной почты. – *Примеч. ред.*

10

Сетевые сценарии

«Подключись, залогинься и отвали»

За последние несколько лет Интернет буквально вырвался на авансцену. Сеть быстро превратилась из простого средства коммуникации, используемого преимущественно учеными и исследователями, в средство информации, которое почти так же вездесуще, как телевидение и телефон. Социологи сравнивают Интернет с периодической печатью по культурному воздействию, а технические комментаторы считают, что все заслуживающие внимания разработки программного обеспечения связаны только с Интернетом. Конечно, только время окончательно рассудит, насколько справедливы такие заявления, но нет сомнений, что Интернет является важным общественным фактором и одной из главных сфер приложения современных программных систем.

Интернет оказался и одной из основных областей применения языка программирования Python. При наличии Python и компьютера, соединяющегося с Интернетом через сокеты, можно написать сценарии Python для чтения и отправки электронной почты в любую точку земного шара, загрузки веб-страниц с удаленных сайтов, передачи файлов по FTP, программирования интерактивных сайтов, синтаксического анализа файлов HTML и XML, а также многого другого, пользуясь при этом лишь модулями Интернета, поставляемыми с Python в качестве стандартных инструментов.

В действительности компании так и действуют по всему свету: работа коммерческих сайтов Yahoo, Infoseek, Hewlett-Packard и многих других опирается на стандартные средства Python. Многие также строят свои сайты и управляют ими с помощью сервера веб-приложений *Zope*, который сам написан на Python и может индивидуально настраиваться с его помощью. Другие используют Python для управления веб-приложениями Java посредством *JPython* (или «Jython») – системы, которая компилирует программы Python в байт-код Java и экспортирует библиотеки Java для использования в сценариях Python.

С ростом Интернета выросла и роль Python в качестве инструмента Интернета. Python хорошо подошел для создания сценариев, работающих с Интернетом, по тем же причинам, которые делают его идеальным в других областях. Его модульная архитектура и короткий цикл разработки хорошо соответствуют напряженным требованиям создания приложений для Интернета. В этой части книги будет показано, что Python не просто поддерживает сценарии Интернета, но и благоприятствует таким качествам, как производительность разработчиков и легкость сопровождения, которые важны для Интернет-проектов всех видов и масштабов.

Темы, относящиеся к сценариям для Интернета

Интернет-программирование охватывает много разных тем, поэтому для облегчения усвоения материала этот предмет был разбит на шесть глав. Данная глава знакомит с

основами Интернета и исследует сокеты – механизм связи, лежащий в основе Интернета. В последующих главах мы перейдем к обсуждению клиента, сервера, создания сайта и более сложных тем.

Каждая глава предполагает знакомство с предыдущей, но обычно можно делать скачки, особенно при наличии некоторого опыта работы с Интернетом. Так как эти главы составляют большую часть книги (около трети), изучаемые темы несколько подробнее описываются ниже.

О чем будет рассказано

Концептуально Интернет можно представить себе состоящим из нескольких функциональных слоев:

Сетевые слои низкого уровня

Механизмы, например транспорт TCP/IP, занимающиеся пересылкой байтов между машинами безотносительно к их смыслу

Сокеты

Интерфейс между программистом и сетью, который действует поверх физических сетевых слоев типа TCP/IP

Протоколы верхнего уровня

Структурированные схемы связи, такие как FTP и e-mail, выполняющиеся поверх сокетов и определяющие форматы сообщений и стандартных адресов

Веб-сценарии на стороне сервера (CGI)

Протоколы «клиент/сервер» высокого уровня для связи между веб-броузерами и веб-серверами, также выполняемые поверх сокетов

Интегрированные среды и инструменты высокого уровня

Системы сторонних разработчиков, такие как Zope и JPython, для решения значительно более широкого круга задач

Основное внимание в этой и следующей за ней главах уделено программированию второго и третьего слоев: сокетов и протоколов более высокого уровня. В этой главе мы начнем с самого низа, изучая модель сетевого программирования с помощью сокетов. Сокеты связаны не только с программированием в Интернете, но они описываются здесь, потому что это их главная роль. Как будет показано, большая часть происходящего в Интернете осуществляется через сокеты, даже когда это не бросается в глаза.

После знакомства с сокетами в следующей главе делается очередной шаг вперед – к интерфейсам Python на стороне клиента для протоколов более высокого уровня, таких как e-mail и FTP, выполняемым поверх сокетов. Оказывается, многое можно сделать, когда у клиента есть лишь Python, и глава 11 представляет образцы сценариев, выполняемых на стороне клиента. В трех главах, следующих далее, представлены сценарии, выполняемые на стороне сервера (программы, выполняемые на компьютере-сервере и обычно запускаемые веб-броузером). Наконец, в главе 15 «Более сложные темы Интернета», последней в этой части, мы кратко познакомимся с инструментами еще более высокого уровня, такими как JPython и Zope.

Попутно будут также задействованы некоторые изученные ранее интерфейсы операционной системы и GUI (процессы, потоки, сигналы и Tkinter) и исследованы некоторые конструктивные варианты и проблемы, возникающие при работе с Интернетом.

В связи с последним заявлением следует сказать еще несколько слов. Сценарии Интернета, наряду с GUI, представляют собой одну из наиболее заманчивых областей применения Python. Как и при работе с GUI, возникает неуловимое, но непосредст-

венное удовлетворение, когда видишь, как программа Python для Интернета распространяет информацию по всему свету. С другой стороны, сетевое программирование по самой своей природе влечет издержки, связанные со скоростью передачи, и ограничения в пользовательских интерфейсах. Некоторые приложения все же лучше не разворачивать в Сети, хотя такая позиция сегодня не в моде. В этой части книги мы честно взглянем в лицо компромиссам, возникающим при работе в Сети.

Многие считают также Интернет чем-то вроде окончательной проверки идеи для инструментов с открытым кодом. Действительно, работа Сети в значительной мере основана на применении большого числа таких инструментов, как Python, Perl, веб-сервер Apache, программа sendmail и Linux. Более того, иногда кажется, что новые инструменты и технологии для веб-программирования появляются быстрее, чем разработчики успевают их освоить.

Положительной стороной Python является нацеленность на интеграцию, делающая его самым подходящим инструментом в таком разнородном мире. В настоящее время программы Python могут устанавливаться как инструменты на стороне клиента или сервера, внедряться в код HTML, использоваться в качестве апплетов и сервлетов в приложениях Java, встраиваться в распределенные системы объектов CORBA и DCOM, интегрироваться с объектами, кодируемыми в XML, и т. д. Говоря более общим языком, основания для использования Python в области Интернета точно такие же, как и в любых других областях: акцент Python на производительности труда, переносимости и интеграции превращает его в идеальное средство для написания программ для Интернета, которые общедоступны, предполагают сопровождение и могут разрабатываться в соответствии со все более сжатыми временными требованиями, характерными для этой области.

О чем мы не будем рассказывать

Теперь, после рассказа о том, что освещается в книге, я должен также отметить, о чем в ней не будет сказано. Как и Tkinter, Интернет представляет собой большую область, и эта часть книги по большей части является знакомством с базовыми понятиями и задачами представления, а не исчерпывающим справочником. Модулей Python для Интернета слишком много, чтобы о каждом из них можно было рассказать в этой книге, но приведенные здесь примеры должны помочь пониманию описаний в руководстве по библиотеке тех модулей, обсуждать которые мы не имеем здесь возможности.

Хочу также отметить, что инструменты более высокого уровня, такие как JPython и Zope, являются самостоятельными большими системами, и для их изучения лучше обратиться к документации, в большей мере ориентированной именно на них. По обеим темам вскоре должны выйти специальные книги, поэтому здесь мы коснемся их лишь вскользь. Кроме того, в данной книге почти ничего не говорится о сетевых уровнях более низкого уровня, таких как TCP/IP. Если вам любопытно, что происходит в Интернете на уровне битов и проводов, обратитесь за подробностями к хорошему учебнику по сетям.

Выполнение примеров этой части книги

Интернет-сценарии обычно предполагают контексты выполнения, отсутствующие в предыдущих примерах этой книги. Это означает, что запуск программ, общающихся через сеть, часто оказывается несколько более сложным. Заранее приведем несколько практических замечаний относительно примеров из этой части книги:

- Для выполнения примеров из этой части книги не требуется загрузки дополнительных пакетов. За исключением главы 15, все будущие примеры основываются на стандартном наборе модулей поддержки Интернета, поставляемых вместе с Python (они устанавливаются в каталог библиотек Python).

- Для запуска большинства примеров этой и последующих глав не требуется сверхсовременного подключения к Сети или учетной записи на веб-сервере; обычно достаточно PC и соединения с Интернетом по коммутируемой линии. Детали конфигурации будут попутно уточняться, но программы на стороне клиента запускаются достаточно просто.
- Не требуется иметь учетную запись на машине с веб-сервером, чтобы *запускать* выполняемые на сервере сценарии, приведенные в последних главах (они могут запускаться любым веб-броузером, подключенным к Сети), но такая учетная запись потребуется для того, чтобы *изменять* эти сценарии.

Когда сценарий Python открывает соединение с Интернетом (с помощью `socket` или модулей протоколов), Python довольствуется любым соединением, которое существует на машине, будь то выделенная линия T1, линия DSL или простой модем. Например, открытие сокета на компьютере с Windows автоматически инициирует при необходимости соединение с провайдером услуг Интернета по коммутируемой линии (на моем переносном компьютере автоматически всплывает диалог Windows для модемного соединения). Иными словами, если у вас есть свой способ соединения с Сетью, то, скорее всего, вы сможете запускать программы данной главы.

Более того, если ваша машина поддерживает сокеты, то, вероятно, она сможет выполнить многие из приведенных здесь примеров, даже если соединение с Интернетом вообще отсутствует. Как мы увидим, имя машины «localhost» или «» обычно означает сам локальный компьютер. Это позволяет тестировать как клиентскую, так и серверную части диалога на одном и том же компьютере, не подключаясь к Сети. Например, на Windows-машине клиенты и серверы могут выполняться локально без выхода в Сеть.

В некоторых последующих примерах предполагается, что на машине сервера выполняется определенный тип сервера (например, FTP, POP, SMTP), но сценарии на стороне клиента работают на любой машине, подключаемой к Интернету, с установленным на ней Python. Примеры серверных сценариев глав 12, 13 и 14 требуют большего: необходимо иметь учетную запись на веб-сервере для кодирования сценариев CGI и отдельно загрузить усовершенствованные системы сторонних разработчиков типа JPython и Zope (или взять их с CD, прилагаемого к данной книге).

Вначале был Грааль

Помимо создания языка Python, Гвидо ван Россум написал на Python броузер для World Wide Web, названный (весьма уместно) Grail (Грааль). Частично Grail разрабатывался для демонстрации возможностей Python. Он позволяет пользователям бродить по Сети, как с помощью Netscape или Internet Explorer, но может также программироваться апплетами Grail – программами Python/Tkinter, загружаемыми с сервера, когда броузер клиента обращается к ним и запускает их. Апплеты Grail действуют во многом аналогично Java-апплетам в более популярных броузерах (подробнее об апплетах в главе 15).

Grail больше не развивается и используется сегодня в основном в исследовательских целях. Но Python по-прежнему пожинает плоды проекта Grail в виде богатого набора инструментов для Интернета. Чтобы написать полноценный веб-броузер, необходимо обеспечить поддержку большого количества протоколов Интернета, и Гвидо оформил их поддержку в виде стандартных библиотечных модулей, поставляемых в настоящее время с языком Python.

Благодаря такому наследству в Python теперь входят стандартная поддержка телеконференций Usenet (NNTP), обработка электронной почты (POP, SMTP, IMAP), пересылка файлов (FTP), веб-страницы и взаимодействие (HTTP, URL,

HTML, CGI) и другие часто используемые протоколы (telnet, Gopher и т. п.). Сценарии Python могут соединяться со всеми этими компонентами Интернета, просто импортируя соответствующие библиотечные модули.

Уже после Grail в библиотеку Python были добавлены дополнительные средства для синтаксического разбора файлов XML, защищенных сокетов OpenSSL и др. Но в значительной мере поддержка Интернета в Python ведет свое происхождение от броузера Grail – еще один пример важности поддержки повторного использования кода в Python. Когда пишется эта книга, Grail все еще можно найти на *http://www.python.org*.

Трубопровод для Интернета

Если только вы не провели последнее десятилетие в пещере, то, должно быть, знаете, что такое Интернет, по крайней мере, с точки зрения пользователя. Функционально это коммуникационная и информационная среда для обмена электронной почтой, просмотра веб-страниц, передачи файлов и т. д. Технически Интернет состоит из целого ряда абстрактных и функциональных слоев – от реальных проводов для передачи битов по всему свету до веб-броузера, получающего эти биты и выводящего их на компьютере как текст, графику или звуки.

В данной книге нас в основном интересует интерфейс между программистом и Интернетом. Он тоже состоит из нескольких слоев: *сокетов*, являющихся программируемыми интерфейсами к соединениям низкого уровня между машинами, и стандартных *протоколов*, которые структурируют переговоры, производящиеся через сокеты. Рассмотрим вначале вкратце каждый из этих слоев, а затем погрузимся в детали программирования.

Слой сокетов

Выражаясь простым языком, сокеты служат программируемым интерфейсом для сетевых соединений между компьютерами. Они также образуют основу и низкоуровневый «водопровод» самого Интернета: все известные протоколы Сети верхнего уровня, такие как FTP, веб-страницы и e-mail, в конечном итоге реализуются через сокеты. Иногда также сокеты называют *конечными пунктами коммуникаций*, так как они служат порталами, через которые программы посылают и принимают байты во время общения.

Для программиста сокеты принимают форму группы библиотечных вызовов. Эти вызовы сокетов умеют пересылать байты между машинами с помощью операций низкого уровня, например TCP – сетевого протокола управления передачей данных. На низком уровне TCP умеет передавать байты, но его не интересует, что означают эти байты. Исходя из целей данной книги мы в основном не будем обращать внимания на способ физической передачи посылаемых в сокеты байтов. Однако для полного понимания сокетов нам потребуются некоторые знания о назначении компьютерам имен.

Идентификаторы машин

Предположим на секунду, что вы хотите поговорить по телефону с кем-то, находящимся на другом конце света. В реальном мире вам, вероятно, потребуется номер телефона этого человека или справочник, в котором этот номер можно найти по его имени. То же самое справедливо для Интернета: прежде чем один компьютер сможет общаться с каким-то другим компьютером в киберпространстве, ему нужно узнать номер или имя другого компьютера.

К счастью, в Интернете есть стандартные способы именования удаленных машин и сервисов, предоставляемых этими машинами. Внутри сценария компьютерная программа, с которой нужно связаться через сокет, идентифицируется с помощью пары значений – имени машины и конкретного номера порта на этой машине:

Имена машин

Имя машины может иметь вид строки чисел, разделенных точками, называемой *IP-адресом* (например, 166.93.218.100), или представляться в более читаемом формате, известном как *доменное имя* (например, *starship.python.net*). При работе с доменными именами они автоматически отображаются в соответствующие цифровые с точками. Это отображение осуществляет сервер доменных имен (DNS-сервер) – программа в Сети, осуществляющая ту же функцию, что и ваша местная телефонная справочная служба.

Номера портов

Номер порта – это просто согласованный числовой идентификатор данного разговора в Сети. Так как компьютеры в Сети могут поддерживать разнообразные сервисы, для указания конкретного разговора на данной машине используются номера портов. Для того чтобы две машины могли общаться через Сеть, при инициации сетевых соединений обе они должны связать сокеты с одним и тем же именем машины и номером порта.

Комбинация из имени машины и номера порта однозначно идентифицирует каждый диалог в Сети. Например, компьютер провайдера услуг Интернета может предоставлять клиентам различные услуги – веб-страницы, telnet, передачу по FTP, электронную почту и т. д. Каждому сервису на машине присвоен уникальный номер порта, на который может посылаться запрос. Для получения веб-страниц с веб-сервера программы должны указывать IP-адрес или доменное имя веб-сервера и номер порта, на котором сервер ждет запросы веб-страниц.

Если такая схема покажется несколько необычной, попробуйте представить себе ее на старомодном языке. Например, для того чтобы поговорить по телефону с кем-то внутри компании, обычно требуется набрать номер телефона компании и дополнительный номер того лица, которое вам нужно. Кроме того, если вы не знаете номера компании, то можете поискать его в телефонном справочнике по названию компании. В Сети все происходит почти так же – имена машин идентифицируют наборы сервисов (как компанию), номера портов идентифицируют отдельные сервисы на конкретной машине (как добавочный номер), а доменные имена отображаются в числа IP серверами доменных имен (как телефонные книги).

Когда программы осуществляют связь через сокеты с другой машиной (или с другими процессами на той же машине) особыми способами, в них не должны использоваться порты с номерами, зарезервированными для стандартных протоколов, – числами в диапазоне 0–1023, но чтобы понять причину этого, нужно сначала обсудить протоколы.

Слой протоколов

Сокеты образуют костяк Интернета, но значительная часть происходящего в Сети программируется с помощью *протоколов*,[1] являющихся моделями сообщений более высокого уровня, выполняемыми поверх сокетов. Если сказать коротко, то протоколы

[1] В некоторых книгах термин *протокол* используется также для ссылки на транспортные схемы более низкого уровня, такие как TCP. В данной книге мы называем протоколами структуры более высокого уровня, создаваемые поверх сокетов; если вас интересует происходящее на более низких уровнях, обратитесь к учебнику по сетям.

Интернета определяют структурированный способ общения через сокеты. Обычно они стандартизуют как форматы сообщений, так и номера портов:

- *Форматы сообщений* обеспечивают структуру для байтов, пересылаемых через сокеты во время обмена данными.
- *Номера портов* служат зарезервированными числовыми идентификаторами используемых сокетов, через которые происходит обмен сообщениями.

«Сырые» сокеты (raw sockets) все еще часто используются во многих системах, но чаще (и обычно проще) связь осуществляется с помощью одного из стандартных протоколов Интернета более высокого уровня.

Правила нумерации портов

Технически номер порта сокета может быть любым 16-битовым целым числом от 0 до 65 535. Однако чтобы облегчить программам поиск стандартных протоколов, порты с номерами 0–1023 зарезервированы и назначены стандартным протоколам верхнего уровня. В табл. 10.1 перечислены порты, зарезервированные для многих стандартных протоколов; каждый из них получает один или более номеров из зарезервированного диапазона.

Таблица 10.1. Номера портов, зарезервированные для стандартных протоколов

Протокол	Стандартная функция	Номер порта	Модуль Python
HTTP	веб-страницы	80	`httplib`
NNTP	телеконференции Usenet	119	`nntplib`
FTP data default	пересылка файлов	20	`ftplib`
FTP control	пересылка файлов	21	`ftplib`
SMTP	отправка e-mail	25	`smtplib`
POP3	получение e-mail	110	`poplib`
IMAP4	получение e-mail	143	`imaplib`
Finger	информационный	79	не определен
Telnet	командные строки	23	`telnetlib`
Gopher	пересылка документов	70	`gopherlib`

Клиенты и серверы

Для программистов сокетов наличие стандартных протоколов означает, что порты с номерами 0–1023 не должны использоваться в сценариях, если только не планируется действительное использование одного из этих протоколов верхнего уровня. Это соответствует стандартам и здравому смыслу. Например, программа telnet может открыть диалог с любой поддерживающей telnet машиной, подключаясь к ее порту 23; если бы не было предустановленных номеров портов, все серверы могли бы устанавливать telnet на разные порты. Аналогично сайты стандартно ждут запросы страниц от броузеров на порту 80; если бы они этого не делали, то для посещения любого сайта в Сети требовалось бы знать и вводить номер порта HTTP.

В результате определения стандартных номеров портов для сервисов в Сети естественно возникает *архитектура клиент/сервер*. С одной стороны есть машины, поддерживающие стандартные протоколы, на которых постоянно выполняется ряд программ, ожидающих запросов на соединение по зарезервированным портам. С другой

стороны находятся машины, которые связываются с этими программами, чтобы воспользоваться экспортируемыми ими сервисами.

Постоянно выполняемую и ждущую запроса программу обычно называют *сервером*, а соединяющуюся с ней программу – *клиентом*. В качестве примера возьмем знакомую модель обзора Сети. Как показано в табл. 10.1, используемый в Сети протокол HTTP позволяет клиентам и серверам общаться через сокеты на порту 80:

Сервер

Машина, которая хранит сайты, обычно выполняет программу веб-сервера, постоянно ожидающую входящие запросы соединения на сокете, связанном с портом 80. Часто сам сервер не занимается ничем другим, кроме постоянного ожидания запросов к порту; обработка запросов передается порожденным процессам или потокам.

Клиенты

Программы, которым нужно поговорить с этим сервером, для инициации соединения задают имя машины сервера и порт 80. Типичными клиентами веб-серверов являются веб-броузеры, например Internet Explorer или Netscape, но любой сценарий может открыть соединение со стороны клиента на порту 80 и получать веб-страницы с сервера.

В целом многие клиенты могут подключаться к серверу через сокеты независимо от того, реализован на нем стандартный протокол или нечто более специфическое для данного приложения. А в некоторых приложениях понятия клиента и сервера становятся расплывчатыми – программы могут обмениваться между собой байтами скорее как равноправные участники (*peers*), а не как главный и подчиненный. Однако в данной книге программы, которые ждут на сокетах, мы обычно называем серверами, а программы, устанавливающие соединения, клиентами. Иногда также мы называем сервером и клиентом машины, на которых выполняются эти программы (например, компьютер, на котором выполняется программа веб-сервера, тоже может быть назван машиной веб-сервером), но это более относится к физической, а не функциональной природе.

Структуры протоколов

Функционально протоколы могут выполнять известную задачу, например чтение электронной почты или передачу сообщения в телеконференцию Usenet, но в конечном счете все сводится к пересылке байтов сообщений через сокеты. Структура байтов этих сообщений зависит от протокола, сокрыта в библиотеке Python и по большей части находится за рамками данной книги, но несколько общих слов помогут развеять таинственность слоя протоколов.

Одни протоколы могут определять содержимое сообщений, пересылаемых через сокеты; другие могут задавать последовательность управляющих сообщений, которыми обмениваются стороны при разговоре. Путем определения правильных схем связи протоколы делают ее более надежной. Они также могут препятствовать появлению блокировок, возникающих, когда машина ждет сообщения, которое никогда не поступит.

Например, протокол FTP предотвращает блокировку путем организации связи по двум сокетам: один из них служит только для управляющих сообщений, а другой – для передачи данных файла. Сервер FTP ждет управляющих сообщений (например, «передай мне файл») на одном порту, а передает данные файла по другому. Клиенты FTP открывают соединения сокетов с управляющим портом машины сервера, посылают запросы и передают или получают данные файлов через сокет, соединенный с портом данных на машине сервера. FTP определяет также стандартные структуры со-

общений, передаваемые между клиентом и сервером. Например, управляющее сообщение запроса файла должно соответствовать стандартному формату.

Библиотечные модули Python для Интернета

Если все это показалось вам ужасно сложным, не унывайте: все детали обрабатываются стандартными модулями Python для протоколов. Например, библиотечный модуль Python `ftplib` управляет установлением связи на уровне сокетов и сообщений, которое определено в протоколе FTP. Сценарии, импортирующие `ftplib`, получают доступ к интерфейсу пересылки файлов по FTP значительно более высокого уровня и могут в значительной мере оставаться в неведении относительно лежащего в основе протокола FTP и сокетов, на которых он выполняется.[1]

В действительности все поддерживаемые протоколы представлены файлами стандартных модулей Python с именами в формате *xxxlib.py*, где *xxx* заменяется именем протокола на нижнем регистре. В последней колонке табл. 10.1 указано имя стандартного модуля для протокола. Например, FTP поддерживается файлом модуля *ftplib.py*. Кроме того, в модулях протоколов имя объекта интерфейса верхнего уровня обычно совпадает с названием протокола. Так, например, чтобы начать сеанс FTP в сценарии Python, нужно выполнить `import ftplib` и передать надлежащие параметры в вызове `ftplib.FTP()`; для telnet нужно создать `telnetlib.Telnet()`.

Помимо модулей реализации протоколов, указанных в табл. 10.1, в стандартной библиотеке Python есть модули для анализа и обработки данных, которые переданы через сокеты или протоколы. В табл. 10.2 перечислены наиболее часто используемые модули из этой категории.

Таблица 10.2. Часто используемые для работы с Интернетом стандартные модули

Модули Python	Применение
`socket`	Поддержка сетевых соединений на низком уровне (TCP/IP, UDP и т. д.)
`cgi`	Поддержка CGI-сценариев на стороне сервера: анализ входного потока, преобразование текста HTML и т. п.
`urllib`	Получение веб-страниц по адресам (URL), кодирование текста URL
`httplib, ftplib, nntplib`	Модули поддержки протоколов HTTP (веб), FTP (пересылка файлов) и NNTP (телеконференции)
`poplib, imaplib, smtplib`	Модули поддержки протоколов POP, IMAP (получение почты) и SMTP (отправка почты)
`telnetlib, gopherlib`	Модули поддержки протоколов telnet и Gopher
`htmllib, sgmllib, xmllib`	Синтаксический анализ содержимого веб-страниц (документы HTML, SGML и XML)
`xdrlib`	Переносимая кодировка двоичных данных (см. также модули `struct` и `socket`)

[1] Так как Python является системой с открытым исходным кодом, можно прочесть исходный код модуля `ftplib`, если вас интересует, как действительно работает используемый протокол. Посмотрите файл *ftplib.py* в стандартном каталоге исходного текста библиотек своей машины. Этот код сложен (он должен форматировать сообщения и управлять двумя сокетами), но, как и другие стандартные модули протоколов Интернета, дает хороший пример низкоуровневого программирования сокетов.

Таблица 10.2 (продолжение)

Модули Python	Применение
rfc822	Синтаксический анализ строк заголовков в стиле e-mail
mhlib, mailbox	Обработка сложных почтовых сообщений и почтовых ящиков
mimetools, mimify	Обработка тела сообщений в стиле MIME
multifile	Чтение сообщений, состоящих из нескольких частей
uu, binhex, base64, binascii, quopri	Кодирование и раскодирование двоичных (или иных) данных, передаваемых как текст
urlparse	Разбор строки URL на составляющие части
SocketServer	Структура для сетевых серверов общего вида
BaseHTTPServer	Базовая реализация сервера HTTP
SimpleHTTPServer, CGIHTTPServer	Специфические модули обработчиков запросов к веб-серверу HTTP
rexec, bastion	Режимы ограничений на выполнение кода

В нескольких последующих главах мы встретимся со многими, хотя и не со всеми из перечисленных в этой таблице модулей. Показанные модули типичны, но, как обычно, обратитесь к стандартному справочному руководству по библиотекам Python за более полными и свежими данными.

О стандартах протоколов

Если необходимы полные сведения о протоколах и портах, то на момент написания этой книги полный список всех портов, зарезервированных для протоколов или зарегистрированных в качестве используемых различными стандартными системами, можно найти поиском по страницам веб, поддерживаемым IETF (Internet Engineering Task Force – оперативной группой инженеров Интернета) и IANA (Internet Assigned Numbers Authority – полномочным комитетом по надзору за номерами, используемыми в Интернете). IETF отвечает за сопровождение протоколов и стандартов веб. IANA является главным координатором назначения уникальных значений параметров для протоколов Интернета. Еще один орган стандартизации, W3 (от WWW), тоже сопровождает соответствующие документы. Подробности смотрите на следующих страницах:

http://www.ietf.org
http://www.iana.org/numbers.html
http://www.isi.edu/in-notes/iana/assignments/port-numbers
http://www.w3.org

Вполне возможно, что в то время, когда у этой книги еще будут читатели, возникнут более свежие хранилища спецификаций стандартных протоколов, но в течение ближайшего времени главным авторитетом будет, по-видимому, служить сайт IETF. Однако если вы соберетесь туда обратиться, то предупреждаем, что детали там, хм... слишком детализированы. Так как модули протоколов Python скрывают большую часть сложных вещей, связанных с сокетами и сообщениями и документированных в стандартах протоколов, обычно не требуется запоминать эти документы, чтобы работать в Сети с помощью Python.

Программирование сокетов

Теперь, когда мы знаем, какую роль играют сокеты в общей картине Интернета, пойдем дальше и посмотрим, какие средства предоставляет Python для программирования сокетов в сценариях Python. В этом разделе показывается, как использовать интерфейс Python к сокетам для осуществления связи в сети на низком уровне. В последующих главах мы, напротив, будем использовать модули протоколов более высокого уровня, скрывающих лежащие в их основе сокеты.

Основным интерфейсом сокетов в Python служит стандартный библиотечный модуль `socket`. Подобно модулю POSIX `os`, модуль Python `socket` служит лишь тонкой оболочкой (интерфейсным слоем) для лежащих в основании вызовов библиотеки C для сокетов. Подобно файлам Python, этот модуль *основывается на объектах*: методы объекта сокета, реализованные в этом модуле, после преобразования данных вызывают соответствующие операции библиотеки C. В модуле `socket` есть также средства для преобразования байтов в стандартный сетевой порядок, создания оболочек объектов сокетов в виде простых файловых объектов и др. Он работает на любой машине, поддерживающей сокеты в стиле BSD – MS Windows, Linux, Unix и так далее, – и таким образом обеспечивает переносимый интерфейс сокетов.

Основы сокетов

Чтобы установить связь между машинами, программы Python импортируют модуль `socket`, создают объект сокета и вызывают методы этого объекта для установления соединения, отправки и получения данных. Методы объекта сокета прямо отображаются в вызовы сокетов библиотеки C. Например, сценарий примера 10.1 реализует программу, которая просто ждет соединения на сокете и отправляет обратно через сокет все, что она через него получает, добавляя префикс `'Echo=>'`.

Пример 10.1. PP2E\Internet\Sockets\echo-server.py

```
############################################################################################
# Сторона сервера: открыть сокет на порту, ждать сообщения клиента, отправить ответ эхом;
# для каждого клиента это простой одноразовый подход ждать/ответить, но при запуске
# этого сценария сервера он входит в бесконечный цикл, ожидая других клиентов;
############################################################################################

from socket import *                            # получить конструктор socket и константы
myHost = ''                                     # машина сервера, '' означает локальный хост
myPort = 50007                                  # ждать на незарезервированном порту

sockobj = socket(AF_INET, SOCK_STREAM)          # создать объект сокета TCP
sockobj.bind((myHost, myPort))                  # связать с номером порта сервера
sockobj.listen(5)                               # ждать, допуская 5 соединений в очереди

while 1:                                        # ждать на порту до конца процесса
    connection, address = sockobj.accept()     # ждать соединения очередного клиента
    print 'Server connected by', address       # соединение является новым сокетом
    while 1:
        data = connection.recv(1024)            # читать следующую строку из сокета клиента
        if not data: break                      # послать клиенту ответ
        connection.send('Echo=>' + data)        # до eof при закрытии сокета
    connection.close()
```

Как уже говорилось, обычно такие программы, которые ожидают входящих соединений, мы называем *серверами*, потому что они предоставляют сервис, доступный на данной машине и порту через Интернет. Программы, подключающиеся к такому сер-

веру для доступа к его сервису, обычно называются *клиентами*. Пример 10.2 показывает простой клиент, реализованный на Python.

Пример 10.2. PP2E\Internet\Sockets\echo-client.py

```
##########################################################################################
# Сторона клиента: с помощью сокетов данные отправляются серверу и выводится ответ сервера
# на каждую строку сообщения; 'localhost' означает, что сервер выполняется на одной машине
# с клиентом, позволяя тестировать клиент и сервер на одной машине; для тестирования
# через Интернет запустите сервер на удаленной машине и установите serverHost или argv[1]
# равным доменному имени или IP-адресу; сокеты Python являются переносимым интерфейсом
# сокетов BSD с методами объектов для стандартных вызовов сокетов;
##########################################################################################

import sys
from socket import *                    # переносимый интерфейс сокетов плюс константы
serverHost = 'localhost'                # имя сервера, например: 'starship.python.net'
serverPort = 50007                      # незарезервированный порт, используемый сервером

message = ['Hello network world']       # текст по умолчанию, посылаемый серверу
if len(sys.argv) > 1:
    serverHost = sys.argv[1]            # сервер в аргументе 1 командной строки
    if len(sys.argv) > 2:               # текст в аргументах командной строки 2..n
        message = sys.argv[2:]          # одно сообщение для каждого заданного аргумента

sockobj = socket(AF_INET, SOCK_STREAM)      # создать объект сокета TCP/IP
sockobj.connect((serverHost, serverPort))   # соединение с сервером и портом

for line in message:
    sockobj.send(line)                  # послать серверу строчку через сокет
    data = sockobj.recv(1024)           # получить строчку от сервера: до 1k
    print 'Client received:', `data`

sockobj.close()                         # закрыть сокет, чтобы послать серверу eof
```

Вызовы сокетов сервером

Прежде чем посмотреть на эти программы в действии, коротко объясним, как клиент и сервер выполняют свои функции. Оба они представляют достаточно простые примеры сценариев сокетов, но иллюстрируют общую схему вызовов, применяемую в большинстве программ, использующих сокеты. В действительности это стереотипный код: большинство программ сокетов обычно выполняет те же вызовы сокетов, что и наши два сценария, поэтому разберем каждую строку из важных мест этих сценариев.

Программы, подобные приведенной в примере 10.1, предоставляющие сервисы другим программам с помощью сокетов, обычно начинают работу с такой последовательности вызовов:

```
sockobj = socket(AF_INET, SOCK_STREAM)
```

Здесь с помощью модуля Python socket создается объект сокета TCP. Имена `AF_INET` и `SOCK_STREAM` принадлежат предопределенным переменным, импортируемым из модуля socket; их совместное применение означает «создать сокет TCP/IP», стандартное средство связи для Интернета. Более точно, `AF_INET` означает протокол адресов IP, а `SOCK_STREAM` означает протокол передачи TCP.

При использовании в этом вызове других имен можно создавать такие объекты, как сокеты UDP без логического соединения (второй параметр `SOCK_DGRAM`) и сокеты доменов Unix на локальной машине (первый параметр `AF_UNIX`), но в данной книге мы этого делать не будем. Смотрите подробности относительно этих и других параметров модуля socket в руководстве по библиотеке Python.

```
sockobj.bind((myHost, myPort))
```

Связывает объект сокета с адресом – для IP-адресов передается имя машины-сервера и номер порта на этой машине. Здесь сервер идентифицирует машину и порт, связанные с сокетом. В серверных программах имя машины обычно задается пустой строкой (« »), что означает машину, на которой выполняется сценарий, а порт указывается как число вне диапазона 0–1023 (зарезервированного для стандартных протоколов, описывавшихся выше). Обратите внимание, что у каждого отдельного поддерживаемого диалога сокетов должен быть свой номер порта. Если попытаться открыть сокет на порту, который уже используется, Python возбудит исключительную ситуацию. Обратите также внимание на вложенные скобки в этом вызове – здесь для сокета с протоколом адресов `AF_INET` мы передаем в `bind` связываемый адрес сокета хост/порт как объект кортежа из двух элементов (для `AF_UNIX` передается строка). Технически `bind` принимает кортеж значений, соответствующий типу создаваемого сокета (см. однако, следующую врезку примечания относительно более старого и не рекомендованного к использованию соглашения по передаче этой функции параметров как отдельных аргументов).

```
sockobj.listen(5)
```

Начинает ожидание входящих клиентских соединений и позволяет помещать в задел (backlog) до пяти ждущих обработки запросов. Передаваемое значение устанавливает количество входящих клиентских запросов, помещаемых операционной системой в очередь перед тем, как начать отклонять новые запросы (что происходит только тогда, когда сервер не успевает обрабатывать запросы и очередь переполняется). Для большинства программ, работающих с сокетами, обычно достаточно значения 5; должно быть задано число не менее 1.

После этого сервер готов к принятию запросов соединения от клиентских программ, выполняющихся на удаленных машинах (или той же самой машине), и входит в бесконечный цикл в ожидании их поступления:

```
connection, address = sockobj.accept()
```

Ждет появления нового запроса клиента о соединении; когда он произойдет, `accept` возвратит новый объект сокета, через который можно передавать данные соединившемуся клиенту и получать их от него. Соединение осуществляет `sockobj`, но связь с клиентом происходит через новый сокет, `connection`. Этот вызов возвращает кортеж из двух элементов – `address` является Интернет-адресом соединившегося клиента. Вызов `accept` может происходить неоднократно, чтобы обслужить несколько соединений клиентов. Поэтому каждый вызов возвращает отдельный новый сокет, через который происходит связь с конкретным клиентом.

Установив соединение с клиентом, мы попадаем в другой цикл, в котором получаем от клиента данные блоками по 1024 байта и эхом отражаем каждый блок обратно клиенту:

```
data = connection.recv(1024)
```

Читает не более 1024 новых байтов из очередного сообщения, посланного клиентом (то есть поступившего из сети), и возвращает их сценарию в виде строки. При завершении клиентом работы возвращается пустая строка: когда клиент закрывает свой конец сокета, возвращается конец файла.

```
connection.send('Echo=>' + data)
```

Отправляет последний полученный блок данных обратно программе клиента, предварив его строкой `'Echo=>'`. Программа клиента получает эти отправленные ей данные через `recv`.

```
connection.close()
```

Закрывает соединение с данным конкретным клиентом.

Завершив разговор с данным конкретным клиентом, сервер возвращается в свой бесконечный цикл и ждет следующего запроса соединения от клиента.

Вызовы сокетов клиентом

С другой стороны, клиентские программы, вроде показанной в примере 10.2, используют более простую последовательность вызовов. Главное, о чем нужно помнить, это необходимость задания клиентом и сервером при открытии своих сокетов одного и того же номера порта и идентификации клиентом машины, на которой выполняется сервер (в наших сценариях сервер и клиент договорились использовать для связи порт с номером 50007 вне диапазона стандартных протоколов):

```
sockobj = socket(AF_INET, SOCK_STREAM)
```

Создает в клиентской программе объект Python для сокета, так же как на сервере.

```
sockobj.connect((serverHost, serverPort))
```

Открывает соединение с машиной и портом, где программа сервера ждет клиентских соединений. В этом месте клиент указывает тот сервис, с которым ему нужно связаться. Клиент может задать имя удаленной машины в виде доменного имени (например, *starship.python.net*) или числового IP-адреса. Можно также задать имя сервера как *localhost*, указав тем самым, что программа сервера выполняется на той же машине, что и клиент; это удобно для отладки серверов без подключения к Сети. И снова номер порта клиента должен в точности соответствовать номеру порта сервера. Обратите внимание на вложенные скобки – так же, как в вызовах `bind` на сервере, мы передаем `connect` адрес хоста/порта сервера в виде объекта-кортежа.

Установив соединение с сервером, клиент попадает в цикл, посылая сообщения построчно и выводя то, что возвращает сервер после передачи каждой строки:

```
sockobj.send(line)
```

Пересылает серверу через сокет очередную строку сообщения.

```
data = sockobj.recv(1024)
```

Читает очередную строку ответа, переданную программой-сервером. Технически читается до 1024 байтов очередной строки ответа, которые возвращаются как строка.

```
sockobj.close()
```

Закрывает соединение с сервером, посылая сигнал «конец файла».

Вот и все. Сервер обменивается одной или несколькими строками текста с каждым подключившимся клиентом. Операционная система обеспечивает поиск удаленных машин, направляя пересылаемые между программами байты через Интернет и (с помощью TCP) обеспечивая доставку сообщений в целости. Для этого требуется еще масса работы – по пути наши строки могут путешествовать по всему свету, переходя из телефонных линий в спутниковые каналы, и т. д. Но при программировании в Python мы можем оставаться в счастливом неведении относительно того, что происходит ниже уровня вызовов сокетов.

В старом коде Python можно увидеть, как адрес сервера `AF_INET` передается в `bind` на стороне сервера и в `connect` на стороне клиента в виде двух разных аргументов, а не кортежа из двух элементов:

```
soc.bind(host,port)      а не soc.bind((host,port))
soc.connect(host,port)   а не soc.connect((host,port))
```

Такой формат с двумя аргументами сейчас не рекомендуется использовать, и только по недосмотру он работал в прежних версиях Python (к несчастью, пример сокетов в руководстве по библиотеке Python тоже содержал вызов в таком формате с двумя аргументами!). Предпочтительным является формат адреса сервера в виде кортежа; вразрез с принятой в Python практикой полной обратной совместимости, в будущих версиях Python останется, вероятно, только один формат.

Локальное выполнение программ с сокетами

Ладно, пусть эти клиент и сервер поработают. Есть два способа запустить эти сценарии – на одной и той же машине или на разных. Чтобы запустить клиент и сервер на одной и той же машине, откройте две консоли командной строки, запустите в одной программу сервера, а в другой несколько раз запускайте клиент. Сервер работает постоянно и отвечает на запросы, которые происходят при каждом запуске сценария клиента в другом окне.

Вот, например, текст, который появляется в окне консоли MS-DOS, в котором я запустил сценарий сервера:

```
C:\...\PP2E\Internet\Sockets>python echo-server.py
Server connected by ('127.0.0.1', 1025)
Server connected by ('127.0.0.1', 1026)
Server connected by ('127.0.0.1', 1027)
```

В выводе указан адрес (IP-имя машины и номер порта) каждого соединившегося клиента. Как и большинство серверов, этот сервер выполняется вечно, ожидая запросов соединения от клиентов. Здесь он получил три запроса, но чтобы понять их значение, нужно показать текст в окне клиента:

```
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world'

C:\...\PP2E\Internet\Sockets>python echo-client.py localhost spam Spam SPAM
Client received: 'Echo=>spam'
Client received: 'Echo=>Spam'
Client received: 'Echo=>SPAM'

C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Shrubbery
Client received: 'Echo=>Shrubbery'
```

Здесь сценарий клиента был запущен три раза, в то время как сценарий сервера постоянно выполнялся в другом окне. Каждый клиент, соединявшийся с сервером, посылал ему сообщение из одной или нескольких строк текста и считывал ответ, возвращаемый сервером – эхо каждой строки текста, отправленной клиентом. И при запуске каждого клиента в окне сервера появлялось новое сообщение о соединении (вот почему там их три).

Важно отметить, что клиенты и сервер выполняются здесь на одной и той же машине (PC с Windows). Сервер и клиент используют один номер порта, но имена машин «» и «localhost» соответственно при ссылке на компьютер, на котором они выполняются. В действительности здесь нет никакого соединения через Интернет. Сокеты хорошо выполняют роль средства связи между программами на одной машине.

Удаленное выполнение программ с сокетами

Чтобы заставить эти сценарии общаться через Интернет, а не в пределах одной машины, необходимо проделать некоторую дополнительную работу для запуска сервера на

другом компьютере. Во-первых, нужно загрузить файл с исходным сервером на удаленную машину, где у вас есть учетная запись и Python. Вот как я выполняю это через FTP (имя вашего сервера и детали интерфейса загрузки могут отличаться, а кроме того, есть другие способы копирования файлов на компьютер, например e-mail, формы для передачи на веб-страницах и т. п.):[1]

```
C:\...\PP2E\Internet\Sockets>ftp starship.python.net
Connected to starship.python.net.
User (starship.python.net:(none)): lutz
331 Password required for lutz.
Password:
230 User lutz logged in.
ftp> put echo-server.py
200 PORT command successful.
150 Opening ASCII mode data connection for echo-server.py.
226 Transfer complete.
ftp: 1322 bytes sent in 0.06Seconds 22.03Kbytes/sec.
ftp> quit
```

После пересылки программы сервера на другой компьютер нужно запустить ее там. Соединитесь с этим компьютером и запустите программу сервера. Обычно я подключаюсь к машине своего сервера через telnet и запускаю программу сервера из командной строки как постоянно выполняющийся процесс.[2] Для запуска сценария сервера в фоновом режиме из оболочек Unix/Linux может использоваться синтаксис с &; можно также сделать сервер непосредственно исполняемым с помощью строки #! и команды *chmod* (подробности в главе 2 «Системные инструменты»). Вот текст, который появляется в окошке моего PC при подключении в сеансе telnet к серверу Linux, на котором у меня есть учетная запись (опущены несколько информационных строк):

```
C:\...\PP2E\Internet\Sockets>telnet starship.python.net
Red Hat Linux release 6.2 (Zoot)
Kernel 2.2.14-5.0smp on a 2-processor i686
login: lutz
Password:
[lutz@starship lutz]$ python echo-server.py &
[1] 4098
```

Теперь, когда сервер ждет соединений через Сеть, снова несколько раз запустите клиент на своем локальном компьютере. На этот раз клиент выполняется на машине, отличной от сервера, поэтому передадим доменное имя или IP-адрес сервера как аргумент командной строки клиента. Сервер по-прежнему использует имя машины «», так как он всегда слушает, на какой бы машине ни выполнялся. Вот что появляется в окне telnet сервера:

```
[lutz@starship lutz]$ Server connected by ('166.93.68.61', 1037)
Server connected by ('166.93.68.61', 1040)
Server connected by ('166.93.68.61', 1043)
Server connected by ('166.93.68.61', 1050)
```

---

[1] Команда FTP стандартно имеется на машинах с Windows и большинстве других. В Windows просто введите ее в окне консоли DOS, чтобы соединиться с сервером FTP (или запустите свою любимую программу FTP); в Linux введите команду FTP в окне xterm. Для подключения к неанонимному FTP-сайту потребуется ввести имя учетной записи и пароль. Для анонимного FTP в качестве имени пользователя укажите «anonymous», а в качестве пароля – свой адрес электронной почты (анонимные FTP-сайты обычно имеют ограничения).

[2] Telnet является стандартной командой на машинах Windows и Linux. В Windows введите ее имя в консоли DOS или в диалоговом окне Start/Run (можно запускать ее и щелчком по значку). Telnet обычно выполняется в собственном окне.

А вот что выводится в окно консоли MS-DOS, когда я запускаю клиент. Сообщение «connected by» появляется в окне telnet сервера каждый раз, когда сценарий клиента запускается в окне клиента:

```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net
Client received: 'Echo=>Hello network world'

C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net ni Ni NI
Client received: 'Echo=>ni'
Client received: 'Echo=>Ni'
Client received: 'Echo=>NI'

C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Shrubbery
Client received: 'Echo=>Shrubbery'

C:\...\PP2E\Internet\Sockets>ping starship.python.net
Pinging starship.python.net [208.185.174.112] with 32 bytes of data:
Reply from 208.185.174.112: bytes=32 time=311ms TTL=246
ctrl-C
C:\...\PP2E\Internet\Sockets>python echo-client.py 208.185.174.112 Does she?
Client received: 'Echo=>Does'
Client received: 'Echo=>she?'
```

Получить IP-адрес машины по доменному имени можно с помощью команды «ping»; чтобы соединиться, клиент может использовать оба формата имени машины. Этот вывод излишне сдержан и скрывает от глаз многое происходящее. Клиент, выполняющийся под Windows на моем переносном компьютере, соединяется с программой сервера, выполняемой на машине под Linux, находящейся, возможно, на расстоянии тысяч миль, и обменивается с ней данными. Это происходит почти так же быстро, как при нахождении клиента и сервера на одной машине; при этом используются одни и те же библиотечные вызовы, и изменяется только имя сервера, передаваемое клиентам.

Практическое использование сокетов

Прежде чем двигаться дальше, следует сказать о трех деталях практического использования сокетов. Во-первых, такие клиент и сервер могут выполняться на любых двух подключенных к Интернету машинах, где установлен Python. Конечно, чтобы запускать клиенты и сервер на разных машинах, необходимы действующее соединение с Интернетом и доступ к той машине, на которой должен быть запущен сервер. При этом не требуется дорогого высокоскоростного соединения – для клиентов достаточно простого модема и доступа к Интернету по коммутируемой линии. Когда открываются сокеты, Python может использовать любой тип соединения – от выделенной линии T1 до модема с коммутируемой линией.

Например, на моем переносном компьютере при запуске клиента или открытии сеанса сервера telnet Windows автоматически дозванивается до провайдера. В примерах этой книги программы сервера, выполняемые удаленно, запускаются на машине с именем *starship.python.net*. Если у вас нет собственной учетной записи на таком сервере, запускайте примеры клиента и сервера на одной машине, как показывалось выше; для этого лишь требуется, чтобы компьютер разрешал использовать сокеты, что бывает почти всегда.

Во-вторых, модуль socket обычно возбуждает исключительную ситуацию при запросе чего-либо недопустимого. Например, неудачной будет попытка подключения к несуществующему серверу (или недоступному серверу, если нет связи с Интернетом):

```
C:\...\PP2E\Internet\Sockets>python echo-client.py www.nonesuch.com hello
Traceback (innermost last):
```

```
  File "echo-client.py", line 24, in ?
    sockobj.connect((serverHost, serverPort))   # соединиться с машиной сервера...
  File "<string>", line 1, in connect
socket.error: (10061, 'winsock error')
```

Наконец, следите за тем, чтобы убить процесс сервера, прежде чем запустить его заново, потому что иначе порт окажется занятым и вы получите другую исключительную ситуацию:

```
[lutz@starship uploads]$ ps -x
  PID TTY      STAT   TIME COMMAND
 5570 pts/0    S      0:00 -bash
 5570 pts/0    S      0:00 -bash
 5633 pts/0    S      0:00 python echo-server.py
 5634 pts/0    R      0:00 ps -x
[lutz@starship uploads]$ python echo-server.py
Traceback (most recent call last):
  File "echo-server.py", line 14, in ?
    sockobj.bind((myHost, myPort))               # связать с номером порта сервера
socket.error: (98, 'Address already in use')
```

В Python 1.5.2 сервер под Linux будет убит, если несколько раз нажать <Ctrl>+<C> (если он был запущен с &, сначала нужно перевести его на передний план с помощью *fg*):

```
[lutz@starship uploads]$ python echo-server.py
ctrl-c
Traceback (most recent call last):
  File "echo-server.py", line 18, in ?
    connection, address = sockobj.accept()       # ждать нового соединения клиента
KeyboardInterrupt
```

Комбинация клавиш <Ctrl>+<C> *не* завершает сервер на моей машине с Windows. Чтобы завершить локальный постоянно выполняющийся процесс сервера под Windows, может потребоваться ввести комбинацию клавиш <Ctrl>+<Alt>+<Delete>, а затем завершить задачу Python, выбрав ее в появившемся окне со списком процессов. В Linux обычно можно завершить работу сервера, если она происходит в другом окне или в фоновом режиме, с помощью команды оболочки `kill -9 pid`, но <Ctrl>+<C> требует меньше нажатий на клавиши.

Порождение параллельных клиентов

Чтобы посмотреть, как сервер справляется с нагрузкой, запустим параллельно восемь экземпляров сценария клиента с помощью сценария примера 10.3 (о деталях модуля `launchmodes`, используемого здесь для порождения клиентов, рассказывается в конце главы 3 «Системные средства параллельного выполнения»).

Пример 10.3. PP2E\Internet\Sockets\testecho.py

```
import sys, string
from PP2E.launchmodes import QuietPortableLauncher

numclients = 8
def start(cmdline): QuietPortableLauncher(cmdline, cmdline)()

# start('echo-server.py')               # локально породить сервер, если он еще не запущен

args = string.join(sys.argv[1:], ' ')   # передать имя сервера при удаленном запуске
for i in range(numclients):
    start('echo-client.py %s' % args)   # запустить 8 клиентов для проверки сервера
```

При запуске этого сценария не передавайте ему аргументов, чтобы связаться с сервером, ждущим соединения на порту 50007 на локальной машине, а передайте действительное имя машины для связи с удаленно выполняющимся сервером. В Windows вывод клиентов отбрасывается, когда они порождаются этим сценарием:

```
C:\...\PP2E\Internet\Sockets>python testecho.py

C:\...\PP2E\Internet\Sockets>python testecho.py starship.python.net
```

Если порождаемые клиенты соединяются с сервером, выполняющимся *локально*, в окне сервера появляются сообщения о соединениях:

```
C:\...\PP2E\Internet\Sockets>python echo-server.py
Server connected by ('127.0.0.1', 1283)
Server connected by ('127.0.0.1', 1284)
Server connected by ('127.0.0.1', 1285)
Server connected by ('127.0.0.1', 1286)
Server connected by ('127.0.0.1', 1287)
Server connected by ('127.0.0.1', 1288)
Server connected by ('127.0.0.1', 1289)
Server connected by ('127.0.0.1', 1290)
```

Если сервер выполняется *удаленно*, сообщения о соединениях клиентов появляются в окне, которое отображает telnet-соединение с удаленным компьютером:

```
[lutz@starship lutz]$ python echo-server.py
Server connected by ('166.93.68.61', 1301)
Server connected by ('166.93.68.61', 1302)
Server connected by ('166.93.68.61', 1308)
Server connected by ('166.93.68.61', 1309)
Server connected by ('166.93.68.61', 1313)
Server connected by ('166.93.68.61', 1314)
Server connected by ('166.93.68.61', 1307)
Server connected by ('166.93.68.61', 1312)
```

Имейте, однако, в виду, что это верно только для наших простых сценариев, поскольку серверу не требуется много времени для ответа на каждый запрос клиента – он может вовремя вернуться в начало внешнего цикла `while`, чтобы обработать следующий входящий клиент. Если бы он не мог этого сделать, то, возможно, потребовалось бы изменить сервер так, чтобы все клиенты обрабатывались *параллельно*, иначе некоторым из них пришлось бы отказать в соединении. Технически подключение следующих клиентов будет неудачным, когда уже есть пять клиентов, ждущих внимания сервера, что определено в вызове `listen` в сервере. В следующем разделе мы увидим, каким образом серверы могут надежно обрабатывать несколько клиентов.

Связь с зарезервированными портами

Важно также помнить, что эти клиент и сервер участвуют в диалоге специфического типа и потому используют порт с номером 50007 – вне диапазона, зарезервированного для стандартных протоколов (0–1023). Однако ничто не мешает клиенту открыть сокет на одном из этих особых портов. Например, следующий код клиента соединяется с программами, ждущими на стандартных портах e-mail, FTP и веб-сервера HTTP на трех разных машинах сервера:

```
C:\...\PP2E\Internet\Sockets>python
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('mail.rmi.net', 110))         # разговор с почтовым сервером POP RMI
```

```
>>> print sock.recv(40)
+OK Cubic Circle's v1.31 1998/05/13 POP3
>>> sock.close()

>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('www.python.org', 21))        # разговор с FTP-сервером Python
>>> print sock.recv(40)
220 python.org FTP server (Version wu-2.
>>> sock.close()

>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('starship.python.net', 80))   # веб-сервер HTTP starship
>>> sock.send('GET /\r\n')                      # получить корневую веб-страницу
7
>>> sock.recv(60)
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">\012<HTM'
>>> sock.recv(60)
'L>\012 <HEAD>\012  <TITLE>Starship Slowly Recovering</TITLE>\012 </HE'
```

Если уметь интерпретировать вывод, возвращаемый серверами этих портов, то можно непосредственно использовать такие сокеты для получения почты, передачи файлов, загрузки веб-страниц и запуска сценариев на сервере. К счастью, нет надобности беспокоиться о деталях происходящего – модули Python `poplib`, `ftplib`, `httplib` и `urllib` предоставляют интерфейсы более высокого уровня для связи с серверами через эти порты. Есть другие модули протоколов Python, которые осуществляют то же самое для других стандартных портов (например, NNTP, telnet и т. д.). С некоторыми из этих модулей протоколов для клиента мы познакомимся в следующей главе.[1]

Между прочим, открывать соединения со стороны клиента по таким зарезервированным портам допускается, но устанавливать собственные *сценарии серверов* для этих портов можно только при наличии особых прав доступа:

```
[lutz@starship uploads]$ python
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.bind(('', 80))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
socket.error: (13, 'Permission denied')
```

Даже если у пользователя есть необходимые права, при выполнении этого кода будет возбуждена исключительная ситуация, если порт уже используется реальным веб-сервером. Компьютеры, используемые как общие серверы, действительно резервируют эти порты.

Обработка нескольких клиентов

Показанные выше программы клиента и сервера `echo` иллюстрируют основы использования сокетов. Но модель сервера страдает довольно существенным недостатком: если несколько клиентов пытаются соединиться с сервером и обработка запроса клиента занимает длительное время, то происходит отказ сервера. Более точно, если тру-

[1] Вам может быть интересно узнать, что последняя часть этого примера, связь через порт 80, представляет в точности то, что делает ваш веб-броузер при серфинге в Сети: переход по ссылкам заставляет его загружать веб-страницы через этот порт. В действительности этот порт составляет главную основу Сети. В главе 12 мы увидим целую среду приложений, основывающуюся на пересылке данных через порт 80 – сценарии CGI для сервера.

доемкость обработки данного запроса не позволит серверу вовремя вернуться в код, проверяющий наличие новых клиентов, сервер не сможет удовлетворить все запросы, и некоторым клиентам будет отказано в соединении.

В реальных программах клиент/сервер код сервера гораздо чаще пишется так, чтобы избежать блокировки новых запросов во время обработки текущего запроса клиента. Вероятно, проще всего достичь этого путем параллельной обработки каждого запроса клиента – в новом процессе, новом потоке или в результате переключения (мультиплексирования) между клиентами вручную в цикле событий. Эта проблема не связана с сокетами как таковыми, и мы уже научились запускать процессы и потоки в главе 3. Но так как эти схемы весьма типичны для программирования серверов, работающих с сокетами, рассмотрим здесь все три способа параллельной обработки запросов клиентов.

Ветвление серверов

Сценарий примера 10.4 действует подобно первоначальному серверу `echo`, но для обработки каждого нового соединения с клиентом ответвляет новый процесс. Так как функция `handleClient` выполняется в новом процессе, функция `dispatcher` может сразу возобновить выполнение своего главного цикла, чтобы обнаружить и обслужить новый поступивший запрос.

Пример 10.4. PP2E\Internet\Sockets\fork-server.py

```
############################################################################################
# На стороне сервера: открыть сокет на порту, ждать сообщения от клиента и отправить
# ответ эхом; ответвляет процесс для обработки каждого соединения с клиентом;
# дочерние процессы используют общие с родителем описатели сокетов;
# ветвление хуже переносимо, чем потоки, – не реализовано в Windows;
############################################################################################

import os, time, sys
from socket import *                                # получить конструктор socket и константы
myHost = ''                                         # машина сервера, '' означает локальный хост
myPort = 50007                                      # слушать на незарезервированном порту

sockobj = socket(AF_INET, SOCK_STREAM)              # создать объект сокета TCP
sockobj.bind((myHost, myPort))                      # связать с номером порта сервера
sockobj.listen(5)                                   # допускается 5 соединений в очереди

def now():                                          # текущее время на сервере
    return time.ctime(time.time())

activeChildren = []
def reapChildren():                                 # убрать мертвые дочерние процессы
    while activeChildren:                           # иначе может переполниться системная таблица
        pid,stat = os.waitpid(0, os.WNOHANG)        # не подвесить, если нет завершения потомков
        if not pid: break
        activeChildren.remove(pid)

def handleClient(connection):                       # дочерний процесс: ответ, выход
    time.sleep(5)                                   # моделирование блокирующих действий
    while 1:                                        # чтение, запись в сокет клиента
        data = connection.recv(1024)                # до eof при закрытии сокета
        if not data: break
        connection.send('Echo=>%s at %s' % (data, now()))
    connection.close()
    os._exit(0)
```

```
def dispatcher():                                   # слушать, пока не будет завершен процесс
    while 1:                                        # ждать следующего соединения,
        connection, address = sockobj.accept()      # передать процессу для обслуживания
        print 'Server connected by', address,
        print 'at', now()
        reapChildren()                              # теперь убрать завершившиеся потомки
        childPid = os.fork()                        # копировать этот процесс
        if childPid == 0:                           # если в дочернем процессе: обработать
            handleClient(connection)
        else:                                       # иначе: принимать следующее соединение
            activeChildren.append(childPid)         # добавить pid к списку активные потомки

dispatcher()
```

Запуск ветвящегося сервера

Некоторые части этого сценария написаны довольно замысловато, и большинство библиотечных вызовов в нем работает только на Unix-подобных платформах (не в Windows). Но прежде чем подробно вникать в детали, запустим наш сервер и обработаем несколько клиентских запросов. Прежде всего, обратите внимание, что для моделирования длительных операций (таких, как обновление базы данных или какая-то пересылка в сети) этот сервер добавляет 5-секундную задержку `time.sleep` в функцию обработки клиентского запроса `handleClient`. После задержки выполняется действие ответа эхом, как и раньше. Это значит, что на этот раз при запуске сервера и клиентов последние получат эхо-ответ не ранее, чем через 5 секунд после отправки запроса серверу.

Чтобы помочь следить за запросами и ответами, сервер выводит *свое* системное время при каждом получении от клиента запроса на соединение и добавляет *свое* системное время к ответу. Клиенты выводят время ответа, полученное с сервера, а не свое собственное – часы на сервере и у клиента могут быть установлены совершенно по-разному, поэтому для того, чтобы складывать яблоки с яблоками, все временные моменты отмечаются по времени сервера. Из-за моделируемой задержки обычно приходится запускать каждый клиент под Windows в собственном окне консоли (на некоторых платформах клиенты висят в заблокированном состоянии, пока не получат свой ответ).

Но самое важное здесь то, что сценарий выполняет на машине сервера один главный родительский процесс, единственной функцией которого является ожидание соединений (в `dispatcher`), плюс один дочерний процесс на каждое активное соединение с клиентом, выполняемый параллельно с главным родительским процессом и другими клиентскими процессами (в `handleClient`). В принципе, сервер не должен захлебнуться при обработке любого количества клиентов. Для проверки запустим сервер удаленно в окне telnet и запустим три клиента локально в трех разных окнах консоли:

```
[server telnet window]
[lutz@starship uploads]$ uname -a
Linux starship …
[lutz@starship uploads]$ python fork-server.py
Server connected by ('38.28.162.194', 1063) at Sun Jun 18 19:37:49 2000
Server connected by ('38.28.162.194', 1064) at Sun Jun 18 19:37:49 2000
Server connected by ('38.28.162.194', 1067) at Sun Jun 18 19:37:50 2000
```

[окно клиента 1]

```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net
Client received: 'Echo=>Hello network world at Sun Jun 18 19:37:54 2000'
```

[окно клиента 2]

```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Bruce
Client received: 'Echo=>Bruce at Sun Jun 18 19:37:54 2000'
```

[окно клиента 3]

```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net The Meaning of Life
Client received: 'Echo=>The at Sun Jun 18 19:37:55 2000'
Client received: 'Echo=>Meaning at Sun Jun 18 19:37:56 2000'
Client received: 'Echo=>of at Sun Jun 18 19:37:56 2000'
Client received: 'Echo=>Life at Sun Jun 18 19:37:56 2000'
```

И снова все моменты времени отмерены по часам машины сервера. Это может быть несколько запутанным, поскольку участвуют четыре окна. На обычном языке тест можно описать так:

1. Начинается удаленное выполнение сервера.
2. Запускаются все три клиента, соединяющиеся с сервером примерно в одно и то же время.
3. На сервере три клиентских запроса запускают три ответвленных дочерних процесса, которые сразу засыпают на пять секунд (изображая занятость чем-то полезным).
4. Каждый клиент ждет ответа сервера, который генерируется через пять секунд после первоначального запроса.

Иными словами, все три клиента обслуживаются в одно и то же время ответвленными процессами, в то время как главный родительский процесс продолжает ждать новых клиентских запросов. Если бы клиенты *не* обрабатывались параллельно, ни один из них не смог бы соединиться до истечения пятисекундной задержки, вызванной обработкой текущего соединенного клиента.

В реальном приложении такая задержка могла оказаться роковой при попытке одновременного подключения нескольких клиентов – сервер застрял бы на операции, которую мы моделируем с помощью `time.sleep`, и не вернулся в главный цикл, чтобы принять новые запросы клиентов. При ответвлении процесса по каждому запросу все клиенты могут обслуживаться параллельно.

Обратите внимание, что сценарий клиента используется прежний (*echo-client.py*), а сервер – другой: клиенты просто посылают свои данные на порт машины и получают их оттуда, не зная того, как их запросы обрабатываются на сервере. Обратите также внимание, что сервер удаленно выполняется на машине под Linux. (Как известно из главы 3, вызов `fork` не поддерживается под Windows в Python на момент написания книги.) Можно запустить этот тест целиком на сервере Linux посредством двух окон telnet. Это работает примерно так же, как при локальном запуске клиентов в окне консоли DOS, но здесь «локальный» означает удаленную машину, с которой вы работаете локально через telnet:

[одно окно telnet]

```
[lutz@starship uploads]$ python fork-server.py &
[1] 3379
Server connected by ('127.0.0.1', 2928) at Sun Jun 18 22:44:50 2000
Server connected by ('127.0.0.1', 2929) at Sun Jun 18 22:45:08 2000
Server connected by ('208.185.174.112', 2930) at Sun Jun 18 22:45:50 2000
```

[другое окно telnet на той же машине]

```
[lutz@starship uploads]$ python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 22:44:55 2000'

[lutz@starship uploads]$ python echo-client.py localhost niNiNI
Client received: 'Echo=>niNiNI at Sun Jun 18 22:45:13 2000'

[lutz@starship uploads]$ python echo-client.py starship.python.net Say no More!
Client received: 'Echo=>Say at Sun Jun 18 22:45:55 2000'
```

```
Client received: 'Echo=>no at Sun Jun 18 22:45:55 2000'
Client received: 'Echo=>More! at Sun Jun 18 22:45:55 2000'
```

Теперь перейдем к ловким приемам. Код сценария сервера, организующий ветвление, достаточно прост, но следует прокомментировать использование некоторых библиотечных средств.

Ветвление процессов

Мы познакомились с `os.fork` в главе 3, однако вспомните, что ветвящиеся процессы в сущности являются копией породившего их процесса и наследуют у родительского процесса описатели файлов и сокетов. Благодаря этому новый дочерний процесс, выполняющий функцию `handleClient`, имеет доступ к сокету соединения, созданному в родительском процессе. Программы узнают о том, что они находятся в ответвленном дочернем процессе, если вызов fork возвращает 0; в противном случае в исходный родительский процесс возвращается ID нового дочернего.

Завершение дочерних процессов

В предшествующих примерах ветвления дочерние процессы обычно вызывали одну из разновидностей `exec` для запуска новой программы в дочернем процессе. Здесь же дочерний процесс просто вызывает функцию в той же программе и завершается с помощью `os._exit`. Здесь необходимо вызывать `os._exit` – если этого не сделать, дочерний процесс продолжает существовать после возврата из `handleClient` и также участвовать в приеме новых клиентских запросов.

На самом деле без вызова exit мы получили бы столько вечных процессов сервера, сколько обслужено запросов – уберите вызов exit и выполните команду оболочки *ps* после запуска нескольких клиентов – вы поймете, что я имею в виду. При наличии этого вызова только один родительский процесс ждет новые запросы. `os._exit` похож на `sys.exit`, но завершает вызвавший его процесс сразу, не производя уборки. Обычно он используется только в дочерних процессах, а `sys.exit` используется во всех остальных случаях.

Удаление зомби

Заметьте, однако, что убедиться в завершении и смерти дочернего процесса не вполне достаточно. В таких системах, как Linux, родительский процесс должен также выполнить системный вызов `wait`, чтобы удалить записи, касающиеся умерших дочерних процессов, из системной таблицы процессов. Если этого не делать, то дочерние процессы выполняться не будут, но будут занимать место в системной таблице процессов. Для серверов, выполняемых длительное время, такие фальшивые записи могут вызвать неприятности.

Такие мертвые, но числящиеся в строю процессы обычно называют «зомби» (zombies): они продолжают использовать системные ресурсы, даже после своей кончины и возврата в операционную систему. Для уборки за ушедшими дочерними процессами наш сервер ведет список `activeChildren`, состоящий из ID всех порожденных им дочерних процессов. При получении нового входящего запроса клиента сервер выполняет `reapChildren`, чтобы вызвать `wait` для умерших дочерних процессов путем выполнения стандартного вызова Python `os.waitpid(0,os.WNOHANG)`.

Вызов `os.waitpid` пытается ожидать завершения дочернего процесса и возвращает ID его процесса и статус завершения. При передаче 0 в качестве первого аргумента ожидается завершение любого дочернего процесса. При передаче `WNOHANG` в качестве второго аргумента вызов ничего не делает, если никакой дочерний процесс не завершится

(то есть вызвавший не блокируется и не приостанавливается). В итоге данный вызов просто запрашивает у операционной системы ID процесса для любого завершившегося дочернего процесса. Если такой процесс есть, возвращенный для него ID удаляется из системной таблицы процессов и из списка `activeChildren` этого сценария.

Чтобы понять, для чего нужны такие сложности, закомментируйте в этом сценарии `reapChildren`, запустите его на сервере, а затем запустите несколько клиентов. На моем сервере Linux команда `ps -f` полного листинга процессов показывает, что все умершие дочерние процессы сохраняются в системной таблице процессов (помечены как `<defunct>`):

```
[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz      3270  3264  0 22:33 pts/1    00:00:00 -bash
lutz      3311  3270  0 22:37 pts/1    00:00:00 python fork-server.py
lutz      3312  3311  0 22:37 pts/1    00:00:00 [python <defunct>]
lutz      3313  3311  0 22:37 pts/1    00:00:00 [python <defunct>]
lutz      3314  3311  0 22:37 pts/1    00:00:00 [python <defunct>]
lutz      3316  3311  0 22:37 pts/1    00:00:00 [python <defunct>]
lutz      3317  3311  0 22:37 pts/1    00:00:00 [python <defunct>]
lutz      3318  3311  0 22:37 pts/1    00:00:00 [python <defunct>]
lutz      3322  3270  0 22:38 pts/1    00:00:00 ps -f
```

Если снова активизировать команду `reapChildren`, записи об умерших дочерних зомби удаляются каждый раз, когда сервер получает новый запрос соединения клиента, путем вызова функции Python `os.waitpid`. Если сервер сильно загружен, может накопиться несколько зомби, но они сохранятся только до получения нового клиентского соединения:

```
[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz      3270  3264  0 22:33 pts/1    00:00:00 -bash
lutz      3340  3270  0 22:41 pts/1    00:00:00 python fork-server.py
lutz      3341  3340  0 22:41 pts/1    00:00:00 [python <defunct>]
lutz      3342  3340  0 22:41 pts/1    00:00:00 [python <defunct>]
lutz      3343  3340  0 22:41 pts/1    00:00:00 [python <defunct>]
lutz      3344  3270  6 22:41 pts/1    00:00:00 ps -f
[lutz@starship uploads]$
Server connected by ('38.28.131.174', 1170) at Sun Jun 18 22:41:43 2000

[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz      3270  3264  0 22:33 pts/1    00:00:00 -bash
lutz      3340  3270  0 22:41 pts/1    00:00:00 python fork-server.py
lutz      3345  3340  0 22:41 pts/1    00:00:00 [python <defunct>]
lutz      3346  3270  0 22:41 pts/1    00:00:00 ps -f
```

Если печатать очень быстро, можно на деле увидеть, как дочерний процесс превращается из реальной выполняющейся программы в зомби. Здесь, например, дочерний процесс, порожденный для обработки нового запроса (ID процесса 11785), при выходе превращается в `<defunct>`. Запись о процессе будет полностью удалена при получении следующего запроса:

```
[lutz@starship uploads]$
Server connected by ('38.28.57.160', 1106) at Mon Jun 19 22:34:39 2000
[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
```

```
lutz     11780 11089  0 22:34 pts/2    00:00:00 python fork-server.py
lutz     11785 11780  0 22:34 pts/2    00:00:00 python fork-server.py
lutz     11786 11089  0 22:34 pts/2    00:00:00 ps -f

[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
lutz     11780 11089  0 22:34 pts/2    00:00:00 python fork-server.py
lutz     11785 11780  0 22:34 pts/2    00:00:00 [python <defunct>]
lutz     11787 11089  0 22:34 pts/2    00:00:00 ps -f
```

Предотвращение появления зомби с помощью обработчиков сигналов

В некоторых системах можно удалять дочерние процессы-зомби путем переустановки обработчика для сигнала SIGCHLD, выставляемого операционной системой по завершении дочернего процесса. Если сценарий Python назначит действие SIG_IGN (игнорировать) в качестве обработчика сигнала SIGCHLD, зомби будут удаляться автоматически и немедленно по завершении дочерних процессов; не требуется, чтобы родительский процесс выполнял вызовы wait для уборки за ними. Благодаря этому такая схема служит более простой альтернативой ручной уборке зомби (на тех платформах, где она поддерживается).

Если вы прочли главу 3, то знаете, что стандартный модуль Python signal позволяет сценариям устанавливать обработчики *сигналов* – программно-генерируемых событий. Если вы не читали эту главу, то вот небольшой пример, который показывает, как это можно использовать для зомби. Программа примера 10.5 устанавливает написанную на Python функцию обработчика сигналов, реагирующую на номер сигнала, вводимый в командной строке.

Пример 10.5. PP2E\Internet\Sockets\signal-demo.py

```
############################################################################################
# Демонстрация модуля Python signal; номер сигнала передается в аргументе
# командной строки, команда оболочки "kill -N pid" посылает этому процессу сигнал;
# например, на моей машине с Linux SIGUSR1=10, SIGUSR2=12, SIGCHLD=17 и
# обработчик SIGCHLD остается действующим, даже если не восстанавливается:
# все остальные обработчики восстанавливаются Python после перехвата, но SIGCHLD
# остается соответственно реализации для платформы; signal работает в Windows,
# но определяет мало типов сигналов; сигналы в целом не очень хорошо переносимы;
############################################################################################

import sys, signal, time

def now():
    return time.ctime(time.time())

def onSignal(signum, stackframe):             # обработчик сигналов python
    print 'Got signal', signum, 'at', now()  # большинство обработчиков продолжает действовать
    if signum == signal.SIGCHLD:              # но не обработчик sigchld
        print 'sigchld caught'
        #signal.signal(signal.SIGCHLD, onSignal)

signum = int(sys.argv[1])
signal.signal(signum, onSignal)                    # установить обработчик сигнала
while 1: signal.pause()                            # спать в ожидании сигналов
```

Для запуска этого сценария просто поместите его в фоновый режим и посылайте ему сигналы, вводя командную строку оболочки в виде kill -номер-сигнала id-процесса. Идентификаторы процессов перечислены в колонке PID результатов выполнения ко-

манды *ps*. Вот как действует этот сценарий, перехватывая сигналы с номерами 10 (зарезервирован для общего использования) и 9 (безусловный сигнал завершения):

```
[lutz@starship uploads]$ python signal-demo.py 10 &
[1] 11297
[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
lutz     11297 11089  0 21:49 pts/2    00:00:00 python signal-demo.py 10
lutz     11298 11089  0 21:49 pts/2    00:00:00 ps -f

[lutz@starship uploads]$ kill -10 11297
Got signal 10 at Mon Jun 19 21:49:27 2000

[lutz@starship uploads]$ kill -10 11297
Got signal 10 at Mon Jun 19 21:49:29 2000

[lutz@starship uploads]$ kill -10 11297
Got signal 10 at Mon Jun 19 21:49:32 2000

[lutz@starship uploads]$ kill -9 11297
[1]+  Killed                  python signal-demo.py 10
```

А теперь сценарий перехватывает сигнал 17, который оказывается SIGCHLD на моем сервере Linux. Номера сигналов зависят от машины, поэтому обычно следует пользоваться их именами, а не номерами. Поведение SIGCHLD тоже может зависеть от платформы (подробности смотрите в руководстве по библиотеке для модуля signal):

```
[lutz@starship uploads]$ python signal-demo.py 17 &
[1] 11320
[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
lutz     11320 11089  0 21:52 pts/2    00:00:00 python signal-demo.py 17
lutz     11321 11089  0 21:52 pts/2    00:00:00 ps -f

[lutz@starship uploads]$ kill -17 11320
Got signal 17 at Mon Jun 19 21:52:24 2000
[lutz@starship uploads] sigchld caught

[lutz@starship uploads]$ kill -17 11320
Got signal 17 at Mon Jun 19 21:52:27 2000
[lutz@starship uploads]$ sigchld caught
```

Теперь, чтобы все это применить для удаления зомби, просто установите для обработчика сигнала SIGCHLD действие SIG_IGN; в системах, где поддерживается такое назначение, дочерние процессы будут вычищены по их завершении. Вариант ветвящегося сервера, показанный в примере 10.6, использует этот прием для управления своими дочерними процессами.

Пример 10.6. PP2E\Internet\Sockets\fork-server-signal.py

```
############################################################################################
# То же, что fork-server.py, но использует модуль Python signal, чтобы избежать сохранения
# дочерних зомби-процессов после их завершения, а не явный цикл перед каждым новым соединением;
# SIG_IGN означает игнорирование и может действовать с сигналом завершения дочернего
# процесса SIG_CHLD не на всех платформах; в Linux socket.accept не может прерываться сигналом;
############################################################################################

import os, time, sys, signal, signal
from socket import *                        # получить конструктор socket и константы
```

```
myHost = ''                                     # машина сервера, '' означает локальный хост
myPort = 50007                                  # ждать на незарезервированном порту

sockobj = socket(AF_INET, SOCK_STREAM)          # создать объект сокета TCP
sockobj.bind((myHost, myPort))                  # связать с номером порта сервера
sockobj.listen(5)                               # до 5 соединений в очереди
signal.signal(signal.SIGCHLD, signal.SIG_IGN)   # избегать дочерних зомби-процессов
def now():                                      # время на машине сервера
    return time.ctime(time.time())

def handleClient(connection):                   # дочерний процесс отвечает, завершается
    time.sleep(5)                               # моделирование блокирующих действий
    while 1:                                    # чтение, запись в сокет клиента
        data = connection.recv(1024)
        if not data: break
        connection.send('Echo=>%s at %s' % (data, now()))
    connection.close()
    os._exit(0)

def dispatcher():                               # слушать, пока не будет завершен процесс
    while 1:                                    # ждать следующего соединения,
        connection, address = sockobj.accept()  # передать процессу для обслуживания
        print 'Server connected by', address,
        print 'at', now()
        childPid = os.fork()                    # копировать этот процесс
        if childPid == 0:                       # если в дочернем процессе: обработать
            handleClient(connection)            # иначе: принимать следующее соединение

dispatcher()
```

Там, где возможно ее применение, такая технология:

- гораздо проще – не нужно вручную следить за дочерними процессами и убирать их;
- более точна – нет временно присутствующих зомби в промежутке между запросами клиентов.

На самом деле обработке зомби здесь посвящена всего одна строка кода: вызов `signal.signal` в начале, устанавливающий обработчик. К сожалению, данная версия еще в *меньшей* степени переносима, чем первая с использованием `os.fork`, поскольку действие сигналов может несколько различаться в зависимости от платформы. Например, на некоторых платформах вообще не разрешается использовать `SIG_IGN` в качестве действия для `SIGCHLD`. Однако в системах Linux этот более простой сервер с ветвлением действует чудесно:

```
[lutz@starship uploads]$
Server connected by ('38.28.57.160', 1166) at Mon Jun 19 22:38:29 2000

[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
lutz     11827 11089  0 22:37 pts/2    00:00:00 python fork-server-signal.py
lutz     11835 11827  0 22:38 pts/2    00:00:00 python fork-server-signal.py
lutz     11836 11089  0 22:38 pts/2    00:00:00 ps -f

[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
lutz     11827 11089  0 22:37 pts/2    00:00:00 python fork-server-signal.py
lutz     11837 11089  0 22:38 pts/2    00:00:00 ps -f
```

Обратите внимание, что в этой версии запись о дочернем процессе исчезает сразу, как только он завершается, даже раньше, чем будет получен новый клиентский запрос. Никаких зомби с пометкой «defunct» не возникает. Еще более знаменательно, что если теперь запустить наш более старый сценарий, порождающий восемь параллельных клиентов (*testecho.py*), соединяющихся с сервером, то все они появляются на сервере при выполнении и немедленно удаляются после завершения:

```
[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
lutz     11827 11089  0 22:37 pts/2    00:00:00 python fork-server-signal.py
lutz     11839 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11840 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11841 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11842 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11843 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11844 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11845 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11846 11827  0 22:39 pts/2    00:00:00 python fork-server-signal.py
lutz     11848 11089  0 22:39 pts/2    00:00:00 ps -f

[lutz@starship uploads]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lutz     11089 11088  0 21:13 pts/2    00:00:00 -bash
lutz     11827 11089  0 22:37 pts/2    00:00:00 python fork-server-signal.py
lutz     11849 11089  0 22:39 pts/2    00:00:00 ps -f
```

Многопоточные серверы

Но никогда так не делайте. Только что описанная модель ветвления в целом хорошо работает на некоторых платформах, но потенциально страдает существенными ограничениями:

Производительность

На некоторых машинах запуск нового процесса обходится довольно дорого в отношении ресурсов времени и памяти.

Переносимость

Ветвление процессов – это инструмент Unix; как уже отмечалось, вызов fork в настоящее время не работает на отличных от Unix платформах, например в Windows.

Сложность

Если вам кажется, что ветвление процессов может быть сложным, то вы правы. Как мы только что видели, ветвление приводит к необходимости решать проблемы, связанные с зомби, – производить зачистку после дочерних процессов, срок жизни которых короче, чем у их родителей.

После прочтения главы 3 вам должно быть известно, что обычным решением этих проблем является использование потоков вместо процессов. Потоки выполняются параллельно и совместно используют глобальную память (то есть память модуля и интерпретатора), но при этом их запуск обычно требует меньших издержек, а работать в настоящее время они могут и на Unix-подобных машинах, и под Microsoft Windows. Кроме того, потоки проще программируются: дочерние потоки тихо умирают по завершении, не оставляя после себя зомби, преследующих сервер.

В примере 10.7 представлена еще одна метаморфоза эхо-сервера, в которой клиентские запросы параллельно обрабатываются в потоках, а не в процессах.

Пример 10.7. PP2E\Internet\Sockets\thread-server.py

```
############################################################################################
# Сторона сервера: открыть сокет на порту, ждать сообщения от клиента и послать эхо-ответ;
# повторяет строки эхом, пока не будет получен eof при закрытии сокета клиентом; порождает поток
# для обработки каждого соединения с клиентом; потоки используют глобальную память совместно
# с главным потоком; это более переносимо, чем fork, которого пока нет в Windows;
############################################################################################

import thread, time
from socket import *                                # получить конструктор socket и константы
myHost = ''                                         # машина сервера, '' означает локальный хост
myPort = 50007                                      # ждать на незарезервированном порту

sockobj = socket(AF_INET, SOCK_STREAM)              # создать объект сокета TCP
sockobj.bind((myHost, myPort))                      # связать с номером порта сервера
sockobj.listen(5)                                   # до 5 соединений в очереди

def now():
    return time.ctime(time.time())                  # текущее время на сервере

def handleClient(connection):                       # в порожденном потоке: ответ
    time.sleep(5)                                   # моделирование блокирующих действий
    while 1:                                        # чтение, запись в сокет клиента
        data = connection.recv(1024)
        if not data: break
        connection.send('Echo=>%s at %s' % (data, now()))
    connection.close()

def dispatcher():                                   # слушать, пока процесс будет завершен
    while 1:                                        # ждать следующего соединения,
        connection, address = sockobj.accept()      # передать процессу для обслуживания
        print 'Server connected by', address,
        print 'at', now()
        thread.start_new(handleClient, (connection,))

dispatcher()
```

Эта функция `dispatcher` передает каждый входящий клиентский запрос в новый порождаемый поток, выполняющий функцию `handleClient`. Благодаря этому данный сервер может одновременно обрабатывать несколько клиентов, а главный цикл диспетчера может быстро вернуться в начало и проверять поступление новых запросов. В результате новым клиентам не будет отказано в обслуживании из-за занятости сервера.

Функционально эта версия аналогична решению `fork` (клиенты обрабатываются параллельно), но может работать на любой машине, поддерживающей потоки, в том числе Windows и Linux. Проверим ее работу в обеих системах. Сначала запустим сервер на машине Linux и клиенты как в Linux, так и в Windows:

[окно 1: серверный процесс, использующий потоки, сервер продолжает принимать соединения клиентов в то время, как потоки обслуживают предшествующие запросы]

```
[lutz@starship uploads]$ /usr/bin/python thread-server.py
Server connected by ('127.0.0.1', 2934) at Sun Jun 18 22:52:52 2000
Server connected by ('38.28.131.174', 1179) at Sun Jun 18 22:53:31 2000
Server connected by ('38.28.131.174', 1182) at Sun Jun 18 22:53:35 2000
Server connected by ('38.28.131.174', 1185) at Sun Jun 18 22:53:37 2000
```

[окно 2: клиент, находящийся на той же машине сервера]

```
[lutz@starship uploads]$ python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 22:52:57 2000'
```

[окно 3: удаленный клиент, PC]
```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net
Client received: 'Echo=>Hello network world at Sun Jun 18 22:53:36 2000'
```

[окно 4: клиент PC]
```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Bruce
Client received: 'Echo=>Bruce at Sun Jun 18 22:53:40 2000'
```

[окно 5: клиент PC]
```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net The Meaning of Life
Client received: 'Echo=>The at Sun Jun 18 22:53:42 2000'
Client received: 'Echo=>Meaning at Sun Jun 18 22:53:42 2000'
Client received: 'Echo=>of at Sun Jun 18 22:53:42 2000'
Client received: 'Echo=>Life at Sun Jun 18 22:53:42 2000'
```

Поскольку этот сервер использует потоки вместо ветвящихся процессов, его можно запускать переносимым образом на Linux- и Windows-компьютерах. Снова запустим его, на этот раз на одной и той же локальной машине Windows вместе с клиентами; и здесь главным является то, что новые клиенты могут быть приняты во время обработки предшествующих клиентов параллельно с другими клиентами и главным потоком (во время 5-секундной искусственной задержки):

[окно 1: сервер на локальном PC]
```
C:\...\PP2E\Internet\Sockets>python thread-server.py
Server connected by ('127.0.0.1', 1186) at Sun Jun 18 23:46:31 2000
Server connected by ('127.0.0.1', 1187) at Sun Jun 18 23:46:33 2000
Server connected by ('127.0.0.1', 1188) at Sun Jun 18 23:46:34 2000
```

[окно 2: клиент, на локальном PC]
```
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 23:46:36 2000'
```

[окно 3: клиент]
```
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Brian
Client received: 'Echo=>Brian at Sun Jun 18 23:46:38 2000'
```

[окно 4: клиент]
```
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Bright side of Life
Client received: 'Echo=>Bright at Sun Jun 18 23:46:39 2000'
Client received: 'Echo=>side at Sun Jun 18 23:46:39 2000'
Client received: 'Echo=>of at Sun Jun 18 23:46:39 2000'
Client received: 'Echo=>Life at Sun Jun 18 23:46:39 2000'
```

Вспомните, что поток тихо завершается при возврате из функции, которую он выполняет; в отличие от версии с ветвлением процессов в функции обработчика клиента не вызывается ничего похожего на `os._exit` (этого и нельзя делать – можно убить все потоки в процессе!). Благодаря этому версия с потоками не только более переносима, но и проста.

То же с помощью классов: структуры для создания серверов

Теперь, когда вы умеете писать серверы с применением ветвления или потоков для обработки клиентов без блокирования входящих запросов, следует сказать, что в библиотеке Python есть стандартные средства, облегчающие этот процесс. В частности, в модуле `SocketServer` определены классы, реализующие все виды ветвящихся и потоковых серверов, которые могут вас заинтересовать. Просто создайте объект сервера нужного импортируемого типа, передав ему объект обработчика с собственным методом обратного вызова, как в примере 10.8.

Пример 10.8. PP2E\Internet\Sockets\class-server.py

```
##########################################################################################
# На сервере: открыть сокет на порту, ждать сообщения от клиента и отправить ответ-эхо;
# в этой версии используется стандартный библиотечный модуль SocketServer;
# SocketServer позволяет создавать простой TCPServer, ThreadingTCPServer, ForkingTCPServer
# и другие и направляет каждый запрос соединения клиента методу handle нового экземпляра
# заданного объекта обработчика запросов; модуль поддерживает также UDP и сокеты домена Unix;
# см. руководство по библиотеке.
##########################################################################################

import SocketServer, time                   # получить объекты сервера сокетов, обработчика
myHost = ''                                 # машина сервера, '' означает локальный хост
myPort = 50007                              # слушать на незарезервированном порту
def now():
    return time.ctime(time.time())

class MyClientHandler(SocketServer.BaseRequestHandler):
    def handle(self):                         # при каждом соединении клиента
        print self.client_address, now()      # показать адрес клиента
        time.sleep(5)                         # моделировать блокирующую операцию
        while 1:                              # self.request есть сокет клиента
            data = self.request.recv(1024)    # читать, писать сокет клиента
            if not data: break
            self.request.send('Echo=>%s at %s' % (data, now()))
        self.request.close()

# создать потоковый сервер, слушать/обрабатывать клиенты непрерывно
myaddr = (myHost, myPort)
server = SocketServer.ThreadingTCPServer(myaddr, MyClientHandler)
server.serve_forever()
```

Этот сервер работает так же, как сервер с потоками, написанный нами вручную в предыдущем разделе, но здесь усилия сосредоточены на реализации сервиса (индивидуальной настройке метода `handle`), а не на деталях работы с потоками. И выполняется он таким же образом – вот результат обработки трех клиентов, созданных вручную, и восьми, порожденных сценарием `testecho` из примера 10.3:

[окно1: сервер, serverHost='localhost' в echo-client.py]

```
C:\...\PP2E\Internet\Sockets>python class-server.py
('127.0.0.1', 1189) Sun Jun 18 23:49:18 2000
('127.0.0.1', 1190) Sun Jun 18 23:49:20 2000
('127.0.0.1', 1191) Sun Jun 18 23:49:22 2000
('127.0.0.1', 1192) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1193) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1194) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1195) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1196) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1197) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1198) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1199) Sun Jun 18 23:49:50 2000
```

[окно2: клиент]

```
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 23:49:23 2000'
```

[окно3: клиент]

```
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Robin
Client received: 'Echo=>Robin at Sun Jun 18 23:49:25 2000'
```

```
[окно4: клиент]
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Brave Sir Robin
Client received: 'Echo=>Brave at Sun Jun 18 23:49:27 2000'
Client received: 'Echo=>Sir at Sun Jun 18 23:49:27 2000'
Client received: 'Echo=>Robin at Sun Jun 18 23:49:27 2000'

C:\...\PP2E\Internet\Sockets>python testecho.py

[окно4: связаться с удаленным сервером – другие времена]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Brave
Sir Robin
Client received: 'Echo=>Brave at Sun Jun 18 23:03:28 2000'
Client received: 'Echo=>Sir at Sun Jun 18 23:03:28 2000'
Client received: 'Echo=>Robin at Sun Jun 18 23:03:29 2000'
```

Чтобы создать сервер, использующий потоки, нужно при создании объекта сервера использовать имя класса `ForkingTCPServer`. Модуль `SocketServer` является более мощным, чем может показаться из данного примера: он поддерживает также синхронные (непараллельные) серверы, UDP и сокеты Unix, и т. д. Подробности в руководстве по библиотеке Python. Кроме того, о средствах реализации серверов в Python сказано в конце главы 15.[1]

Мультиплексирование серверов с помощью select

К настоящему времени мы узнали, как одновременно обрабатывать несколько клиентов с помощью ветвления процессов и порождения потоков, и рассмотрели библиотечный класс, охватывающий обе эти схемы. При том и другом подходе обработчики клиентов выполняются одновременно друг с другом и с главным циклом, продолжающим ожидать новые входящие запросы. Так как все эти задачи выполняются параллельно (то есть в одно и то же время), сервер не блокируется при получении новых запросов или работе долго выполняющегося обработчика клиентов.

Однако технически потоки и процессы на самом деле не выполняются одновременно, если только у вас на машине нет очень большого количества ЦП. В действительности операционная система проделывает фокус – она делит вычислительную мощность процессора между всеми активными задачами, выполняя часть одной, затем часть другой и т. д. Кажется, что все задачи выполняются параллельно, но это происходит только потому, что операционная система переключается между выполнением разных задач так быстро, что обычно это незаметно. Такой процесс переключения между задачами, осуществляемый операционной системой, иногда называют квантованием времени (time-slicing). Более общим его названием является *мультиплексирование* (*multiplexing*).

При порождении потоков и процессов мы рассчитываем, что жонглировать активными задачами будет операционная система, но нет причин, по которым этим не мог бы заниматься также сценарий Python. Например, сценарий может разделять задачи на несколько этапов – выполнить этап одной задачи, затем другой и т. д., пока все они не будут завершены. Чтобы самостоятельно осуществлять мультиплексирование, сценарий должен лишь уметь распределять свое внимание среди нескольких активных задач.

Серверы могут с помощью этой техники осуществить еще один способ одновременной обработки клиентов, при котором не требуются ни потоки, ни ветвление. Мульти-

[1] Между прочим, с Python поставляются также библиотечные средства, позволяющие в нескольких строках кода Python реализовать полнофункциональный сервер HTTP (веб), умеющий выполнять сценарии CGI. Эти средства будут рассмотрены в главе 15.

плексируя соединения клиентов и главного диспетчера с помощью системного вызова select, можно обрабатывать клиенты и принимать новые соединения параллельно (или близко к тому, избегая задержек) в *одном цикле событий*. Такие серверы иногда называются *асинхронными*, поскольку они обслуживают клиенты рывками по мере их готовности для связи. В асинхронных серверах один главный цикл, выполняемый в одном процессе и потоке, решает каждый раз, которому из клиентов должно быть уделено внимание. Запросы клиентов и главный диспетчер получают небольшой квант внимания сервера, если они готовы к общению.

Большинство чудес в такой структуре сервера обеспечивается вызовом select операционной системы, который доступен в Python через стандартный модуль select. Приблизительно действие select заключается в том, что его просят следить за списком источников входных данных, выходных данных и исключительных ситуаций, а он сообщает, какие из источников готовы к обработке. Можно заставить его просто опрашивать все источники, чтобы находить те, которые готовы, ждать готовности источников в течение некоторого предельного времени или ждать неограниченно готовности к обработке одного или нескольких источников.

При любом своем использовании select позволяет направлять внимание на сокеты, которые готовы к связи, чтобы избежать блокировки при обращении к тем, которые не готовы. Это означает, что когда источники, передаваемые select, являются сокетами, можно иметь уверенность, что такие вызовы сокетов, как accept, recv и send, не заблокируют (остановят) сервер при применении к объектам, возвращенным select. Благодаря этому сервер с единственным циклом, но использующий select, не должен застрять при связи с каким-то одним клиентом или в ожидании новых, в то время как остальные клиенты обделены вниманием сервера.

Эхо-сервер на базе select

Посмотрим, как можно организовать это в коде. Сценарий примера 10.9 реализует еще один эхо-сервер, который может обрабатывать несколько клиентов, не открывая новых процессов или потоков.

Пример 10.9. PP2E\Internet\Sockets\select-server.py

```
############################################################################################
# Сервер: параллельная обработка нескольких клиентов с помощью select. Модуль select применен
# для мультиплексирования группы сокетов: главных сокетов, принимающих новые клиентские
# соединения, и входных сокетов, соединенных с принятыми клиентами; select может принять
# дополнительный 4-й аргумент: 0, чтобы опрашивать, n.m, чтобы ждать n.m сек., если опущен,
# ждать готовности любого сокета к обработке.
############################################################################################

import sys, time
from select import select
from socket import socket, AF_INET, SOCK_STREAM
def now(): return time.ctime(time.time())

myHost = ''                                     # машина сервера, '' означает локальный хост
myPort = 50007                                  # слушать на незарезервированном порту
if len(sys.argv) == 3:                          # можно задать хост/порт в командной строке
    myHost, myPort = sys.argv[1:]
numPortSocks = 2                                # количество портов для соединений с клиентами

# создать главные сокеты для приема новых запросов клиентов
mainsocks, readsocks, writesocks = [], [], []
for i in range(numPortSocks):
    portsock = socket(AF_INET, SOCK_STREAM)     # создать объект сокета TCP/IP
```

```
    portsock.bind((myHost, myPort))          # связать его с сервером и портом
    portsock.listen(5)                       # слушать, допуская до 5 соединений в очереди
    mainsocks.append(portsock)               # добавить для идентификации в главный список
    readsocks.append(portsock)               # добавить в список источников select
    myPort = myPort + 1                      # привязка к последовательным портам

# цикл событий: слушать и мультиплексировать, пока жив процесс сервера
print 'select-server loop starting'
while 1:
    #print readsocks
    readables, writeables, exceptions = select(readsocks, writesocks, [])
    for sockobj in readables:
        if sockobj in mainsocks:                         # для готовых входных сокетов
            # сокет порта: принять нового клиента
            newsock, address = sockobj.accept()          # принятие не должно блокировать
            print 'Connect:', address, id(newsock)       # newsock является новым сокетом
            readsocks.append(newsock)                    # добавить к списку select, ждать
        else:
            # сокет клиента: читать новую строку
            data = sockobj.recv(1024)                    # recv не должен блокировать
            print '\tgot', data, 'on', id(sockobj)
            if not data:                                 # если закрыт клиентом
                sockobj.close()                          # закрыть и удалить из
                readsocks.remove(sockobj)                # списка, иначе снова select
            else:
                # может блокировать: для записи тоже нужно select
                sockobj.send('Echo=>%s at %s' % (data, now()))
```

Основу этого сценария составляет большой цикл событий `while`, в котором вызывается `select`, чтобы узнать, какие сокеты готовы к обработке (в том числе главные сокеты, на которых могут соединяться клиенты, и открытые соединения с клиентами). Затем все такие готовые сокеты перебираются в цикле, при этом принимаются соединения на главных сокетах портов и производится чтение или эхо-вывод на клиентских сокетах, готовых для ввода. Вызовы `accept` и `recv` в этом коде гарантированно не блокируют процесс сервера после возврата из `select`; благодаря этому сервер может быстро вернуться в начало цикла и обработать вновь поступившие клиентские запросы и ввод данных уже подключенными клиентами. В итоге все новые запросы и клиенты обслуживаются псевдопараллельным образом.

Чтобы действовал этот процесс, сервер добавляет подключенный сокет каждого клиента в список `readables`, передаваемый в `select`, и просто ждет, когда сокет появится во входном списке, который им возвращается. Для иллюстрации количество портов, на которых этот сервер слушает клиентов, больше одного – это порты 50007 и 50008 в наших примерах. Так как эти главные сокеты портов также опрашиваются `select`, запросы соединения по любому порту могут быть приняты без блокирования уже соединенных клиентов или новых запросов соединения, появляющихся на другом порту. Вызов `select` возвращает те сокеты из списка `readables`, которые готовы к обработке, – как главные сокеты портов, так и сокеты, соединенные с обрабатываемыми в данный момент клиентами.

Запуск сервера, основанного на select

Запустим этот сервер локально и посмотрим, как он работает (клиент и сервер могут запускаться на разных машинах, как в предыдущих примерах). Сначала предположим, что этот сервер уже работает в одном окне, и запустим несколько клиентов для связи с ним. Следующий код показывает диалог в двух таких клиентских окнах под Windows (в консолях MS-DOS). Первый клиент просто дважды запускает сценарий

echo-client для связи с сервером, а второй запускает сценарий testecho, порождающий восемь программ echo-client, выполняющихся параллельно. Как и ранее, сервер просто отражает эхом любой текст, отправленный клиентом. Заметьте, что во втором окне клиента в действительности выполняется сценарий с именем echo-client-50008, который соединяется с сокетом второго порта на сервере; он такой же, как echo-client, но имеет другой номер порта (к сожалению, первоначальный сценарий не позволяет передавать ему имя порта):

```
[окно клиента 1]
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Aug 13 22:52:01 2000'

C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Aug 13 22:52:03 2000'

[окно клиента 2]
C:\...\PP2E\Internet\Sockets>python echo-client-50008.py localhost Sir Lancelot
Client received: 'Echo=>Sir at Sun Aug 13 22:52:57 2000'
Client received: 'Echo=>Lancelot at Sun Aug 13 22:52:57 2000'

C:\...\PP2E\Internet\Sockets>python testecho.py
```

Следующий раздел с кодом показывает, какого рода взаимодействие и вывод происходят в окне, в котором был запущен сервер. Первые три соединения приходят от запущенных echo-client; остальные – результат восьми программ, порожденных testecho во втором окне клиента. Обратите внимание, что для testecho новые клиентские соединения и ввод данных клиента мультиплексируются вместе. Если внимательно изучить вывод, можно заметить перекрытие их по времени, потому что все операции управляются единственным циклом событий на сервере.[1] Заметьте также, что когда клиент закрывает сокет, сервер получает пустую строку. Мы следим за тем, чтобы сразу закрыть и удалить такие сокеты, иначе они будут без нужды снова и снова попадать в список select в главном цикле:

```
[окно сервера]
C:\...\PP2E\Internet\Sockets>python select-server.py
select-server loop starting
Connect: ('127.0.0.1', 1175) 7965520
        got Hello network world on 7965520
        got  on 7965520
Connect: ('127.0.0.1', 1176) 7964288
        got Hello network world on 7964288
        got  on 7964288
Connect: ('127.0.0.1', 1177) 7963920
        got Sir on 7963920
        got Lancelot on 7963920
        got  on 7963920

[результаты testecho]
Connect: ('127.0.0.1', 1178) 7965216
        got Hello network world on 7965216
        got  on 7965216
Connect: ('127.0.0.1', 1179) 7963968
Connect: ('127.0.0.1', 1180) 7965424
        got Hello network world on 7963968
```

[1] Трассировка вывода сервера может выглядеть по-разному при каждом его запуске. Клиенты и новые соединения перемежаются почти случайным образом из-за различий в синхронизации на хостах.

```
Connect: ('127.0.0.1', 1181) 7962976
        got Hello network world on 7965424
        got  on 7963968
        got Hello network world on 7962976
        got  on 7965424
        got  on 7962976
Connect: ('127.0.0.1', 1182) 7963648
        got Hello network world on 7963648
        got  on 7963648
Connect: ('127.0.0.1', 1183) 7966640
        got Hello network world on 7966640
        got  on 7966640
Connect: ('127.0.0.1', 1184) 7966496
        got Hello network world on 7966496
        got  on 7966496
Connect: ('127.0.0.1', 1185) 7965888
        got Hello network world on 7965888
        got  on 7965888
```

Тонкий, но важный момент: делать вызов `time.sleep` для моделирования долгой задачи на этом сервере неразумно – так как все клиенты обрабатываются в одном и том же цикле, задержка остановит их все (и расстроит весь смысл мультиплексирующего сервера). Прежде чем двинуться дальше, сделаем еще несколько замечаний:

Особенности вызова select

Формально в `select` передается три списка выбираемых объектов (входные источники, выходные источники и источники исключительных ситуаций), а также необязательный тайм-аут. Значением аргумента тайм-аута может быть действительное значение *ожидания* в секундах (для задания долей секунды используются числа с плавающей точкой), значение ноль, задающее опрос с немедленным возвратом, или оно может быть опущено, что определяет ожидание готовности по крайней мере одного объекта (как сделано выше в нашем сценарии). Вызов возвращает тройку готовых объектов – подмножеств первых трех аргументов, причем все или некоторые из них могут быть пустыми, если тайм-аут произошел раньше, чем источники стали готовы.

Переносимость select

Вызов `select` в Windows работает только с сокетами, но в Unix и Macintosh также с такими объектами, как файлы и каналы. Конечно, для серверов, работающих в Интернете, сокеты являются основным интересующим нас инструментом.

Неблокирующие сокеты

Благодаря `select` такие вызовы сокетов, как `accept` и `recv`, не блокируют вызвавшего, но есть также возможность сделать сокеты Python неблокирующими в целом. С помощью метода `setblocking` объектов сокетов они устанавливаются в блокирующий или неблокирующий режим. Например, после вызова `sock.setblocking(flag)` сокет `sock` устанавливается в неблокирующий режим, если флаг равен нулю, и в блокирующий режим в противном случае. Все сокеты изначально открываются в блокирующем режиме, поэтому вызовы сокетов всегда могут заставить вызвавшего ждать.

Но при нахождении в неблокирующем режиме возбуждается исключительная ситуация `socket.error`, когда вызов `recv` не находит данных или вызов `send` не может немедленно передать данные. Сценарий может перехватить эту исключительную ситуацию, чтобы определить, готов ли сокет к обработке. В блокирующем режиме эти вызовы всегда осуществляют блокировку, пока не смогут продолжить работу.

Конечно, обработка запроса клиента может отнюдь не ограничиваться пересылкой данных (запросы могут потребовать длительных расчетов), поэтому неблокирующие сокеты не гарантируют отсутствие задержки на сервере в целом. Они лишь предоставляют еще один способ кодирования мультиплексных серверов. Подобно `select`, они лучше подходят для случаев, когда запросы клиентов могут быть обслужены быстро.

Структура, предоставляемая модулем asyncore

Если вас заинтересовало использование `select`, то, вероятно, вам будет интересно и получить модуль *asyncore.py* из стандартной библиотеки Python. Он реализует модель обратного вызова, основанную на классах, в которой обратные вызовы для ввода и вывода отправляются методам класса из готового кода цикла событий `select`. Благодаря этому можно строить серверы без потоков и ветвлений. Подробнее об этом инструменте мы узнаем в конце главы 15.

Выбор схемы сервера

Так в каких же случаях для создания сервера следует использовать `select`, а не потоки или ветвление? Конечно, в каждом приложении свои потребности, но обычно считается, что серверы, основанные на вызове `select`, очень хорошо работают, когда транзакции клиента относительно короткие. Если они не короткие, потоки или ветвление могут оказаться более удачным способом распределить обработку среди нескольких клиентов. Потоки и ветвление особенно полезно применять, если помимо вызовов сокетов клиентам требуется длительная обработка.

Важно помнить, что схемы, основанные на `select` (и неблокирующих сокетах), не вполне защищены от блокирования. Например, в представленном выше примере вызов `send`, который отражает эхом текст клиенту, тоже может оказаться блокирующим и задержать работу всего сервера. Можно было бы преодолеть эту опасность блокирования, применяя `select` для проверки готовности к операции вывода перед попыткой выполнить ее (например, использовать список `writesocks` и добавить еще один цикл для отправки ответов готовым выходным сокетам), но это существенно уменьшило бы ясность программы.

Однако в целом, если нельзя разделить обработку клиентского запроса таким способом, чтобы можно было его мультиплексировать с другими запросами и не заблокировать цикл сервера, `select` может оказаться не лучшим способом построения сервера. Более того, `select` оказывается также более сложным, чем порождение процессов или потоков, поскольку требуется вручную передавать управление всем участвующим задачам (сравните, например, версии этого сервера с потоками и с `select`, даже без отбора по записи). Как всегда, степень этой сложности зависит от конкретного приложения.

Простой файловый сервер на Python

Настало время для более практического кода. Завершим эту главу, применив некоторые из этих относящихся к сокетам идей к несколько более полезной задаче, чем эхо-отражение текста. В примере 10.10 реализована логика как сервера, так и клиента, необходимая для передачи запрошенного файла с сервера на машину клиента прямо через сокеты.

Этот сценарий реализует простую систему загрузки файлов с сервера. Один его экземпляр выполняется на машине, где находятся загружаемые файлы (сервере), а другой – на машине, куда должны копироваться файлы (клиенте). Аргументы командной строки указывают сценарию, в каком качестве он должен выполняться, а также мо-

гут задавать имя машины сервера и номер порта, через который должна производиться связь. Экземпляр сервера может отвечать на любое количество запросов файлов клиентами на порту, который он слушает, так как обслуживает каждый запрос в отдельном потоке.

Пример 10.10. PP2E\Internet\Sockets\getfile.py

```
############################################################################################################
# Реализация логики клиента и сервера для передачи произвольного файла от сервера клиенту
# через сокет; использован простой протокол с управляющей информацией вместо отдельных сокетов
# для управления и данных (как в ftp), посылает каждый клиентский запрос в поток обработчика
# и организует цикл для поблочной передачи всего файла; см. транспортную схему более высокого
# уровня в примерах ftplib;
############################################################################################################

import sys, os, thread, time
from socket import *
def now(): return time.ctime(time.time())

blksz = 1024
defaultHost = 'localhost'
defaultPort = 50001

helptext = """
Usage...
server=> getfile.py  -mode server            [-port nnn] [-host hhh|localhost]
client=> getfile.py [-mode client] -file fff [-port nnn] [-host hhh|localhost]
"""

def parsecommandline():
    dict = {}                                # поместить в словарь для облегчения поиска
    args = sys.argv[1:]                      # пропустить имя программы в начале аргументов
    while len(args) >= 2:                    # пример: dict['-mode'] = 'server'
        dict[args[0]] = args[1]
        args = args[2:]
    return dict

def client(host, port, filename):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.connect((host, port))
    sock.send(filename + '\n')             # отправить имя удаленного файла вместе с каталогом
    dropdir = os.path.split(filename)[1]   # имя файла в конце каталога
    file = open(dropdir, 'wb')             # создать локальный файл в текущем каталоге
    while 1:
        data = sock.recv(blksz)              # получать единовременно не более 1 Кбайт
        if not data: break                   # до закрытия сервером
        file.write(data)                     # сохранить данные в локальном файле
    sock.close()
    file.close()
    print 'Client got', filename, 'at', now()

def serverthread(clientsock):
    sockfile = clientsock.makefile('r')      # обернуть сокет в дублирующий объект файла
    filename = sockfile.readline()[:-1]      # получить имя файла без конца строки
    try:
        file = open(filename, 'rb')
        while 1:
            bytes = file.read(blksz)         # читать/писать по 1 Кбайт
            if not bytes: break              # до полной передачи файла
            sent = clientsock.send(bytes)
```

```
            assert sent == len(bytes)
    except:
        print 'Error downloading file on server:', filename
    clientsock.close()

def server(host, port):
    serversock = socket(AF_INET, SOCK_STREAM)    # слушать на сокете tcp/ip
    serversock.bind((host, port))                # обслуживать клиенты в потоках
    serversock.listen(5)
    while 1:
        clientsock, clientaddr = serversock.accept()
        print 'Server connected by', clientaddr, 'at', now()
        thread.start_new_thread(serverthread, (clientsock,))

def main(args):
    host = args.get('-host', defaultHost)        # использовать args или значения по умолчанию
    port = int(args.get('-port', defaultPort))   # строка в argv
    if args.get('-mode') == 'server':            # None, если нет -mode: клиент
        if host == 'localhost': host = ''        # иначе – отказ при удаленной работе
        server(host, port)
    elif args.get('-file'):                      # в режиме клиента нужен -file
        client(host, port, args['-file'])
    else:
        print helptext

if __name__ == '__main__':
    args = parsecommandline()
    main(args)
```

В этом сценарии нет ничего особенного в сравнении с уже встречавшимися примерами. В зависимости от переданных аргументов командной строки он вызывает одну из двух функций:

- Функция `server` направляет все поступающие клиентские запросы в потоки, передающие байты запрошенного файла.
- Функция `client` посылает серверу имя файла и сохраняет полученные от него байты в локальном файле с таким же именем.

Наибольшая новизна заключается в протоколе между клиентом и сервером: клиент начинает разговор с сервером путем отправки ему строки с именем файла, оканчивающейся символом конца строки и содержащей путь к файлу на сервере. На сервере порожденный поток извлекает имя запрошенного файла, читая сокет клиента, затем открывает запрошенный файл и отправляет его клиенту по частям.

Выполнение файл-сервера и клиентов

Так как сервер использует для обработки клиентов потоки, протестировать сервер и клиент можно на одной и той же машине Windows. Сначала запустим экземпляр сервера, и пока он работает, запустим на той же машине два экземпляра клиента:

[окно сервера, localhost]

```
C:\...\PP2E\Internet\Sockets>python getfile.py -mode server
Server connected by ('127.0.0.1', 1089) at Thu Mar 16 11:54:21 2000
Server connected by ('127.0.0.1', 1090) at Thu Mar 16 11:54:37 2000
```

[окно клиента, localhost]

```
C:\...\Internet\Sockets>ls
class-server.py   echo.out.txt      testdir           thread-server.py
echo-client.py    fork-server.py    testecho.py
```

```
echo-server.py    getfile.py        testechowait.py

C:\...\Internet\Sockets>python getfile.py -file testdir\python15.lib -port 50001
Client got testdir\python15.lib at Thu Mar 16 11:54:21 2000

C:\...\Internet\Sockets>python getfile.py -file testdir\textfile
Client got testdir\textfile at Thu Mar 16 11:54:37 2000
```

Клиенты запускаются в том каталоге, в который нужно поместить загружаемые файлы – код экземпляра клиента обрезает имя пути на сервере при создании имени локального файла. Здесь «загрузка» просто копирует запрошенные файлы в локальный родительский каталог (команда DOS *fc* сравнивает содержимое файлов):

```
C:\...\Internet\Sockets>ls
class-server.py   echo.out.txt      python15.lib      testechowait.py
echo-client.py    fork-server.py    testdir           textfile
echo-server.py    getfile.py        testecho.py       thread-server.py

C:\...\Internet\Sockets>fc /B python1.lib testdir\python15.lib
Comparing files python15.lib and testdir\python15.lib
FC: no differences encountered

C:\...\Internet\Sockets>fc /B textfile testdir\textfile
Comparing files textfile and testdir\textfile
FC: no differences encountered
```

Как обычно, можно запускать сервер и клиенты на разных машинах. Ниже сценарий запускается как сервер на удаленной машине Linux и как несколько клиентов на локальном Windows-компьютере (в некоторые командные строки добавлены переносы, чтобы уместить их на странице). Обратите внимание на появившееся различие времени машин сервера и клиентов – они получаются от часов с разных машин, которые могут иметь произвольный сдвиг между собой:

[окно telnet сервера: первым сообщением является запрос python15.lib в окне клиента 1]

```
[lutz@starship lutz]$ python getfile.py -mode server
Server connected by ('166.93.216.248', 1185) at Thu Mar 16 16:02:07 2000
Server connected by ('166.93.216.248', 1187) at Thu Mar 16 16:03:24 2000
Server connected by ('166.93.216.248', 1189) at Thu Mar 16 16:03:52 2000
Server connected by ('166.93.216.248', 1191) at Thu Mar 16 16:04:09 2000
Server connected by ('166.93.216.248', 1193) at Thu Mar 16 16:04:38 2000
```

[окно клиента 1: запущенный первым выполняется в потоке, в то время как запросы других клиентов делаются в окне 2 и обрабатываются другими потоками]

```
C:\...\Internet\Sockets>python getfile.py -mode client -host starship.python.net
                                                -port 50001 -file python15.lib
Client got python15.lib at Thu Mar 16 14:07:37 2000

C:\...\Internet\Sockets>fc /B python15.lib testdir\python15.lib
Comparing files python15.lib and testdir\python15.lib
FC: no differences encountered
```

[окно клиента 2: запросы делаются во время загрузки по запросу в окне клиента 1]

```
C:\...\Internet\Sockets>python getfile.py -host starship.python.net -file textfile
Client got textfile at Thu Mar 16 14:02:29 2000

C:\...\Internet\Sockets>python getfile.py -host starship.python.net -file textfile
Client got textfile at Thu Mar 16 14:04:11 2000

C:\...\Internet\Sockets>python getfile.py -host starship.python.net -file textfile
Client got textfile at Thu Mar 16 14:04:21 2000
C:\...\Internet\Sockets>python getfile.py -host starship.python.net -file index.html
```

```
Client got index.html at Thu Mar 16 14:06:22 2000

C:\...\Internet\Sockets>fc textfile testdir\textfile
Comparing files textfile and testdir\textfile
FC: no differences encountered
```

Замечание, касающееся безопасности: экземпляр кода сервера готов отправить любой находящийся на сервере файл, имя которого получено от клиента, если сервер выполняется под именем пользователя, имеющего право чтения запрошенного файла. Если вас волнует проблема защиты некоторых своих файлов на сервере, следует добавить логику, запрещающую загрузку защищенных файлов. Пока это оставляется читателю в качестве упражнения, но в главе 12 такие проверки имен файлов будут реализованы в утилите загрузки `getfile`.[1]

Придание сокетам внешнего вида файлов

В иллюстративных целях `getfile` с помощью метода объекта сокета `makefile` создает оболочку сокета в виде объекта, похожего на файл. После создания такой оболочки можно осуществлять чтение и запись сокета посредством обычных методов файлов; в `getfile` используется вызов файла `readline` для чтения строки имени файла, отправленной клиентом.

В данном примере строгой необходимости в этом нет – можно было бы прочесть эту строку и с помощью вызова сокета `recv`. Однако в целом метод `makefile` оказывается удобным, когда требуется передать сокет интерфейсу, который ожидает файл.

Например, в модуле `pickle` методы `load` и `dump` предполагают работу с объектом, интерфейс которого сходен с файловым (например, методы `read` и `write`), но не требуют физического файла. Передача в pickler сокета TCP/IP, обернутого с помощью `makefile`, позволяет передавать через Интернет сериализованные объекты Python. Подробности, касающиеся интерфейсов сериализации объектов, смотрите в главе 16 «Базы данных и постоянное хранение».

В более общем случае любой компонент, предполагающий протокол с методом типа файла, с готовностью примет объект, созданный с помощью метода сокета `makefile`. Такие интерфейсы могут также принимать строки, для которых создана оболочка с использованием встроенного метода `StringIO`, и объект любого другого вида, если он поддерживает такие же вызовы методов, как и встроенные объекты файлов. Как всегда в Python, код пишется согласно *протоколам* – интерфейсам объектов, а не в соответствии с конкретными типами данных.

Добавление графического интерфейса пользователя

Вы могли заметить, что на протяжении всей этой главы мы жили в мире командных строк – наши клиенты и серверы сокетов запускались из простых оболочек DOS или

[1] Там мы встретимся еще с тремя программами `getfile`, прежде чем распрощаться со сценариями для Интернета. В следующей главе сценарий *getfile.py* использует не прямое обращение к сокетам, а получает файлы с помощью интерфейса FTP более высокого уровня, а сценарии *http-getfile* получают файлы по протоколу HTTP. В главе 12 представлен сценарий *getfile.cgi*, передающий содержимое файла через порт HTTP в ответ на запрос, производимый в веб-броузере (файлы посылаются как вывод сценария CGI). Все четыре схемы загрузки, представленные в этой книге, в конечном счете, используют сокеты, но явным это использование является только в данной версии загрузки.

Linux. Однако ничто не мешает нам добавить в некоторые из этих сценариев красивый пользовательский интерфейс «point-and-click» («укажи-и-щелкни»): GUI и сетевые сценарии не являются взаимно исключающими технологиями. На самом деле они могут оказаться достаточно привлекательными при правильном совместном использовании.

Например, нетрудно реализовать с помощью Tkinter простой GUI для клиентской части сценария `getfile`, с которым мы только что познакомились. Такой инструмент, выполняясь на клиентской машине, может просто выводить всплывающее окно с графическими элементами `Entry` для ввода нужных имени файла, сервера и т. д. После ввода параметров загрузки интерфейс пользователя может импортировать и вызвать функцию `getfile.client` с соответствующими аргументами либо построить и выполнить необходимую командную строку *getfile.py* с помощью таких инструментов, как `os.system`, `os.fork`, `thread` и т. д.

Использование фреймов и командных строк

Для большей конкретности рассмотрим очень бегло несколько простых сценариев, добавляющих интерфейс Tkinter к клиентской стороне программы `getfile`. Первый из них (см. пример 10.11) создает диалоговое окно для ввода данных о сервере, порте и имени файла, а потом строит соответствующую команду `getfile` и выполняет ее с помощью `os.system`.

Пример 10.11. PP2E\Internet\Sockets\getfilegui-1.py

```
############################################################################################
# Запуск сценария getfile в режиме клиента из простого GUI Tkinter; можно также os.fork+exec,
# os.spawnv (см. Launcher); windows: заменить 'python' на 'start', если отсутствует в пути;
############################################################################################

import sys, os
from Tkinter import *
from tkMessageBox import showinfo

def onReturnKey():
    cmdline = ('python getfile.py -mode client -file %s -port %s -host %s' %
                      (content['File'].get(),
                       content['Port'].get(),
                       content['Server'].get()))
    os.system(cmdline)
    showinfo('getfilegui-1', 'Download complete')

box = Frame(Tk())
box.pack(expand=YES, fill=X)
lcol, rcol = Frame(box), Frame(box)
lcol.pack(side=LEFT)
rcol.pack(side=RIGHT, expand=Y, fill=X)

labels = ['Server', 'Port', 'File']
content = {}
for label in labels:
    Label(lcol, text=label).pack(side=TOP)
    entry = Entry(rcol)
    entry.pack(side=TOP, expand=YES, fill=X)
    content[label] = entry

box.master.title('getfilegui-1')
box.master.bind('<Return>', (lambda event: onReturnKey()))
mainloop()
```

При выполнении этого сценария создается форма, показанная на рис. 10.1. Нажатие клавиши <Enter> запускает экземпляр программы getfile в режиме клиента; когда сгенерированная командная строка getfile завершается, всплывает окно подтверждения, показанное на рис. 10.2.

Рис. 10.1. getfilegui-1 в действии

Рис. 10.2. Окно подтверждения getfilegui-1

Использование сеток и вызовов функций

В первом сценарии пользовательского интерфейса (пример 10.11) для структурирования формы ввода использован менеджер геометрии pack и фреймы, а клиент getfile выполняется как самостоятельная программа. Для расположения элементов на форме так же просто можно использовать менеджер grid и импортировать и вызвать функцию, реализующую логику клиента, а не запускать программу. Это демонстрируется примером 10.12.

Пример 10.12. PP2E\Internet\Sockets\getfilegui-2.py

```
##########################################################################################
# То же, но с сетками и импортом плюс вызов вместо упаковки и командной строки;
# непосредственный вызов функции обычно быстрее, чем выполнение файла;
##########################################################################################

import getfile
from Tkinter import *
from tkMessageBox import showinfo

def onSubmit():
    getfile.client(content['Server'].get(),
                   int(content['Port'].get()),
                   content['File'].get())
    showinfo('getfilegui-2', 'Download complete')

box    = Tk()
labels = ['Server', 'Port', 'File']
rownum  = 0
content = {}
for label in labels:
    Label(box, text=label).grid(col=0, row=rownum)
    entry = Entry(box)
    entry.grid(col=1, row=rownum, sticky=E+W)
    content[label] = entry
    rownum = rownum + 1

box.columnconfigure(0, weight=0)   # сделать расширяемым
box.columnconfigure(1, weight=1)
Button(text='Submit', command=onSubmit).grid(row=rownum, col=0, columnspan=2)

box.title('getfilegui-2')
box.bind('<Return>', (lambda event: onSubmit()))
mainloop()
```

В этой версии создается аналогичное окно (рис. 10.3), в нижнюю часть которого добавлена кнопка, выполняющая то же, что нажатие клавиши <Enter> – она запускает процедуру клиента `getfile`. Вообще говоря, импорт и вызов функций (как здесь) происходит быстрее, чем выполнение командных строк, особенно если производится многократно. Сценарий `getfile` позволяет использовать его любым способом – как программу или библиотеку функций.

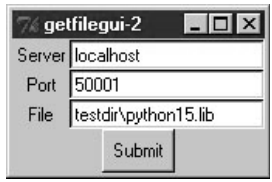


Рис. 10.3. getfilegui-2 в действии

Многократно используемый класс формы

Если у вас такие же склонности, как у меня, то писать весь код формы GUI для этих двух сценариев покажется вам утомительным, как для упаковки, так и для сеток. Для меня это показалось настолько скучным, что я решил написать класс структуры формы общего назначения, показанный в примере 10.13, который выполняет большую часть черновой работы создания структуры элементов GUI.

Пример 10.13. PP2E\Internet\Sockets\form.py

```
# многократно используемый класс формы, применяемый в getfilegui (и др.)

from Tkinter import *
entrysize = 40

class Form:                                          # добавить немодальное окно формы
    def __init__(self, labels, parent=None):         # передать список меток полей
        box = Frame(parent)
        box.pack(expand=YES, fill=X)
        rows = Frame(box, bd=2, relief=GROOVE)       # в окне есть ряды, кнопка
        lcol = Frame(rows)                           # в строке есть левая и правая колонки
        rcol = Frame(rows)                           # кнопка или клавиша <return>
        rows.pack(side=TOP, expand=Y, fill=X)        # вызывают метод onSubmit
        lcol.pack(side=LEFT)
        rcol.pack(side=RIGHT, expand=Y, fill=X)
        self.content = {}
        for label in labels:
            Label(lcol, text=label).pack(side=TOP)
            entry = Entry(rcol, width=entrysize)
            entry.pack(side=TOP, expand=YES, fill=X)
            self.content[label] = entry
        Button(box, text='Cancel', command=self.onCancel).pack(side=RIGHT)
        Button(box, text='Submit', command=self.onSubmit).pack(side=RIGHT)
        box.master.bind('<Return>', (lambda event, self=self: self.onSubmit()))

    def onSubmit(self):                              # переопределить этот метод
        for key in self.content.keys():              # данные, введенные пользователем
            print key, '\t=>\t', self.content[key].get()   # self.content[k]

    def onCancel(self):                              # переопределить при необходимости
        Tk().quit()                                  # по умолчанию завершить
```

```
class DynamicForm(Form):
    def __init__(self, labels=None):
        import string
        labels = string.split(raw_input('Enter field names: '))
        Form.__init__(self, labels)
    def onSubmit(self):
        print 'Field values...'
        Form.onSubmit(self)
        self.onCancel()

if __name__ == '__main__':
    import sys
    if len(sys.argv) == 1:
        Form(['Name', 'Age', 'Job'])       # готовые поля, остаются после передачи
    else:
        DynamicForm()                      # входные поля, исчезают после передачи
    mainloop()
```

При самостоятельном выполнении этого модуля запускается код самотестирования, находящийся в конце. При запуске без аргументов (и двойном щелчке в менеджере файлов Windows) самопроверка генерирует форму с готовыми полями, показанную на рис. 10.4, и выводит значения полей при нажатии клавиши <Enter> или щелчке по кнопке Submit:

```
C:\...\PP2E\Internet\Sockets>python form.py
Job     =>      Educator, Entertainer
Age     =>      38
Name    =>      Bob
```

При задании аргумента в командной строке код самотестирования в модуле класса формы предлагает ввести произвольную группу имен полей для формы: при желании поля могут строиться динамически. Рис. 10.5 показывает форму для ввода, построенную в результате приведенного ниже диалога в консоли. Имена полей могут быть взяты и из командной строки, но в таких простых проверках `raw_input` действует столь же хорошо. В этом режиме GUI исчезает после первой передачи данных, потому что так определено в `DynamicForm.onSubmit`:

```
C:\...\PP2E\Internet\Sockets>python form.py -
Enter field names: Name Email Web Locale
Field values...
Email   =>      lutz@rmi.net
Locale  =>      Colorado
Web     =>      http://rmi.net/~lutz
Name    =>      mel
```

Рис. 10.4. Тест формы, готовые поля

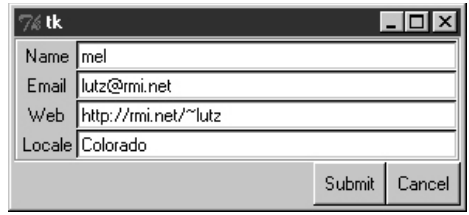


Рис. 10.5. Проверка формы с динамическими полями

Последнее, но немаловажное замечание. В примере 10.14 снова показан интерфейс пользователя `getfile`, на этот раз построенный с помощью многократно используемого класса структуры формы. Необходимо лишь заполнить список меток на форме и предоставить свой метод обратного вызова `onSubmit`. Все действия по построению формы совершаются «бесплатно» в результате импорта многократно используемого суперкласса `Form`.

Пример 10.14. PP2E\Internet\Sockets\getfilegui.py

```
##########################################################################################################
# Запуск клиента getfile с многократно используемым классом формы gui; выполняет os.chdir
# в нужный локальный каталог, если он указан (getfile записывает в текущий каталог);
# доделать: использовать потоки, показать состояние загрузки и вывод getfile;
##########################################################################################################

from form import Form
from Tkinter import Tk, mainloop
from tkMessageBox import showinfo
import getfile, os

class GetfileForm(Form):
    def __init__(self, oneshot=0):
        root = Tk()
        root.title('getfilegui')
        labels = ['Server Name', 'Port Number', 'File Name', 'Local Dir?']
        Form.__init__(self, labels, root)
        self.oneshot = oneshot
    def onSubmit(self):
        Form.onSubmit(self)
        localdir   = self.content['Local Dir?'].get()
        portnumber = self.content['Port Number'].get()
        servername = self.content['Server Name'].get()
        filename   = self.content['File Name'].get()
        if localdir:
            os.chdir(localdir)
        portnumber = int(portnumber)
        getfile.client(servername, portnumber, filename)
        showinfo('getfilegui', 'Download complete')
        if self.oneshot: Tk().quit()  # else stay in last localdir

if __name__ == '__main__':
    GetfileForm()
    mainloop()
```

Импортированный здесь класс структуры формы может быть использован любой программой, которой требуется ввод данных в виде формы; при использовании в данном сценарии получается интерфейс пользователя типа того, который показан на рис. 10.6 для работы в Windows (и аналогичный на других платформах).

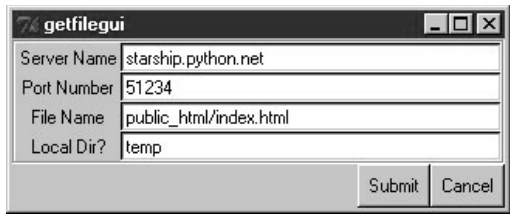


Рис. 10.6. getfilegui в действии

Нажатие кнопки Submit или клавиши <Enter> на этой форме, как и раньше, заставляет сценарий getfilegui вызвать импортированную функцию клиентской части getfile.client. Однако на сей раз сначала также происходит переход в указанный в форме локальный каталог, в который должен быть записан полученный файл (getfile записывает файл в текущий рабочий каталог, каким бы он ни был при вызове сценария). Как обычно, с помощью этого интерфейса можно соединяться с серверами, выполняемыми локально на той же машине или удаленно. Вот пример диалогов, происходящих в том и другом режимах:

```
[связь с локальным сервером]
C:\...\PP2E\Internet\Sockets>python getfilegui.py
Port Number     =>      50001
Local Dir?      =>      temp
Server Name     =>      localhost
File Name       =>      testdir\python15.lib
Client got testdir\python15.lib at Tue Aug 15 22:32:34 2000

[связь с локальным сервером]
[lutz@starship lutz]$ /usr/bin/python getfile.py -mode server -port 51234
Server connected by ('38.28.130.229', 1111) at Tue Aug 15 21:48:13 2000

C:\...\PP2E\Internet\Sockets>python getfilegui.py
Port Number     =>      51234
Local Dir?      =>      temp
Server Name     =>      starship.python.net
File Name       =>      public_html/index.html
Client got public_html/index.html at Tue Aug 15 22:42:06 2000
```

Здесь стоит сделать одно предупреждение: GUI, по существу, мертв, пока происходит загрузка (даже перерисовка экрана не обрабатывается – попробуйте заслонить окно и снова открыть его, и вы поймете, что я имею в виду). Положение можно улучшить, запустив загрузку в потоке, но пока мы не увидим в следующей главе, как это делается, следует считать это предварительным рассмотрением проблемы.

В завершение несколько последних замечаний. Во-первых, я должен отметить, что сценарии этой главы используют приемы использования Tkinter, которые мы уже видели раньше и не станем здесь подробно рассматривать в интересах экономии места; советы по реализации можно найти в главах этой книги, посвященных GUI.

Имейте также в виду, что все эти интерфейсы лишь *добавляют* GUI поверх существующих сценариев, повторно используя их код; таким способом можно снабдить GUI любой инструмент командной строки, сделав его более привлекательным и дружественным пользователю. Например, в следующей главе мы познакомимся с более удобным основанным на Tkinter интерфейсом пользователя на стороне клиента для чтения и отправки электронной почты через сокеты (PyMailGui), который в значительной мере лишь добавляет GUI к средствам обработки почты. Вообще говоря, GUI часто могут быть добавлены к программам почти что задним числом. Хотя степень разделения интерфейса пользователя и базовой логики может быть различной в каждой программе, отделение одного от другого облегчает возможность сосредоточиться на каждом из них.

И наконец теперь, когда я показал, как строить интерфейсы пользователя поверх сценария getfile из этой главы, должен также сказать, что в действительности они не столь полезны, как может показаться. В частности, клиенты getfile могут общаться только с теми машинами, на которых выполняется сервер getfile. В следующей главе мы откроем для себя еще один способ загрузки файлов с сервера, FTP, который также основывается на сокетах, но предоставляет интерфейс более высокого уровня и досту-

пен в качестве стандартного сервиса на многих машинах в Сети. Обычно не требуется запускать индивидуально разработанный сервер для передачи файлов через FTP, как мы это делали с `getfile`. В действительности сценарии с интерфейсом пользователя этой главы можно легко изменить так, чтобы получить нужный файл с помощью инструментов FTP, имеющихся в Python, а не модуля `getfile`. Но я не стану сейчас все рассказывать, а просто предложу продолжить чтение.

Последовательные порты в Windows

Сокеты, главный предмет этой главы, служат интерфейсом программиста к сетевым соединениям в сценариях Python. Как было показано, они позволяют писать сценарии, обменивающиеся данными с компьютерами, расположенными в произвольном месте сети, и образуют становой хребет Интернета и Веб.

Однако если вы ищете средство более низкого уровня для связи с устройствами в целом, для вас может представить интерес тема интерфейсов Python к последовательным портам. Это не имеет полного отношения к сценариям для Интернета и применимо только на машинах Windows, однако достаточно близко по духу и достаточно часто обсуждается в Сети, чтобы быть кратко рассмотренным здесь.

Последовательные порты в Windows называют «COM-портами» (не путать с многокомпонентной моделью объектов COM) и обозначают «COM1», «COM2» и т. д. Используя интерфейсы к этим портам, сценарии могут вести обмен данными с такими устройствами, как мышь, модем и целым рядом других последовательных устройств. Интерфейсы последовательных портов применяются также для связи с устройствами, подключаемыми через инфракрасные порты (например, карманными компьютерами и удаленными модемами). Часто для доступа к таким устройствам могут применяться другие способы более высокого уровня (например, пакет PyRite для доступа к базам данных Palm Pilot или RAS для работы с модемами), но интерфейсы последовательных портов позволяют сценариям вмешиваться в потоки необработанных данных и реализовывать собственные протоколы устройств.

Есть не менее трех способов посылать и принимать данные через последовательные порты в сценариях Python – находящийся в общественном владении пакет расширения C, известный как Serial, интерфейс объекта фирменного COM-сервера MSComm, опубликованный Microsoft, и низкоуровневый вызов файлового API CreateFile, экспортируемый пакетом расширений Python Windows, ссылки на который есть на *http://www.python.org*.

К сожалению, нет возможности подробно обсуждать их в данной книге. Дополнительные сведения можно найти в книге O'Reilly «Python Programming on Win32», содержащей целый раздел, посвященный связи через последовательные порты. Для получения самых свежих сведений по этой теме следует использовать поисковые средства на сайте Python.

11

Сценарии на стороне клиента

«Свяжись со мной!»

В предыдущей главе мы познакомились с основами Интернета и исследовали *сокеты* – механизм связи, лежащий в основе передачи байтов через Сеть. В данной главе мы перейдем на один уровень выше в иерархии инкапсуляции и направим внимание на те инструменты Python, которые поддерживают интерфейсы клиента стандартных протоколов Интернета.

В начале предыдущей главы мы кратко обсудили Интернет-протоколы верхнего уровня, и если этот материал был пропущен при первом чтении, стоит к нему вернуться. Вкратце, протоколы определяют структуру обмена информацией, происходящего при выполнении большинства знакомых нам задач Интернета – чтения электронной почты, передачи файлов по FTP, загрузки веб-страниц и т. п.

В основе своей все эти диалоги протоколов осуществляются через сокеты с использованием фиксированных и стандартных структур сообщений и портов, поэтому в известном смысле данная глава является развитием предыдущей. Но, как будет показано, модули протоколов Python скрывают большую часть деталей – сценариям обычно необходимо иметь дело только с простыми объектами и методами, а Python автоматизирует логику сокетов и сообщений, требуемую протоколом.

В этой главе мы сосредоточимся на модулях Python протоколов FTP и электронной почты, а попутно взглянем и на некоторые другие (новостей NNTP, веб-страниц HTTP и т. д.). Все используемые в примерах инструменты есть в стандартной библиотеке Python и поставляются вместе с системой Python. Все имеющиеся здесь примеры предназначены для выполнения на *клиентской* стороне сетевого соединения – эти сценарии соединяются с уже работающим сервером, которому передают запросы, и могут выполняться на обычном PC. В следующей главе мы перейдем к изучению сценариев, которые, напротив, предназначены для выполнения на стороне сервера. А пока мы займемся клиентом.

Передача файлов по Сети

Как было показано в предыдущей главе, сокеты активно используются в Сети. В частности, пример `getfile` в конце той главы позволял передавать между машинами целые файлы. Однако на практике многое из происходящего в Сети обеспечивается протоколами верхнего уровня. Протоколы выполняются поверх сокетов, но скрывают значительную часть сложностей сетевых сценариев, которые мы видели в приведенных примерах.

FTP (File Transfer Protocol, протокол передачи файлов) – один из наиболее часто используемых протоколов Интернета. Он определяет модель взаимодействия более вы-

сокого уровня, в основе которой лежит обмен командными строками и содержимым файлов через сокеты. FTP может решать те же задачи, что и сценарий `getfile` предыдущей главы, но с более простым и стандартным интерфейсом – FTP позволяет запрашивать файлы с любой машины сервера, которая поддерживает FTP, не требуя, чтобы на ней выполнялся наш специализированный сценарий `getfile`. FTP позволяет также выполнять более сложные операции, такие как загрузка файлов на сервер, получение содержимого удаленного каталога и др.

В действительности FTP выполняется поверх *двух* сокетов: один из них служит для передачи управляющих команд между клиентом и сервером (порт 21), а другой – для передачи байтов. Благодаря использованию модели с двумя сокетами FTP устраняет возможность взаимной блокировки (то есть передача в сокетах данных не блокирует диалога в управляющих сокетах). Однако в конечном итоге вспомогательный модуль Python `ftplib` позволяет загружать файлы на удаленный сервер и с него посредством FTP, не прибегая к чистым вызовам сокетов и не занимаясь деталями протокола FTP.

FTP: загрузка Python с помощью Python

Поскольку интерфейс Python к FTP очень прост, перейдем сразу к практическому примеру. Сценарий примера 11.1 автоматически загружает и компилирует Python с помощью Python. Нет, это не рекурсивное мысленное упражнение о курице и яйце – для выполнения этой программы Python должен быть уже установлен. Более конкретно, этот сценарий Python осуществляет следующее:

1. Загружает дистрибутив исходного кода Python через FTP
2. Распаковывает дистрибутив и компилирует его в исполняемый файл интерпретатора Python

Часть, касающаяся загрузки, работает на любой машине, где есть Python и сокеты; код распаковки и компиляции написан так, что предполагает наличие Unix-образной среды компиляции, но может быть подправлен для работы на других платформах.

Пример 11.1. PP2E\Internet\Ftp\getpython.py

```
#!/usr/local/bin/python
#########################################################################################
# Сценарий Python для загрузки и компиляции исходного кода Python. Использует ftplib--
# обработчик протокола ftp, основанный на сокетах. Ftp действует с 2 сокетами
# (одним для данных, другим для управления - на портах 20 и 21) и налагает форматы на текст
# сообщений, но модуль Python ftplib скрывает большую часть деталей этого протокола.
#########################################################################################

import os
from ftplib import FTP                    # инструменты ftp, основанные на сокетах
Version = '1.5'                           # версия, которую нужно загрузить
tarname = 'python%s.tar.gz' % Version     # имя удаленного/локального файла

print 'Connecting...'
localfile  = open(tarname, 'wb')          # куда записать загруженный файл
connection = FTP('ftp.python.org')        # соединиться с ftp-сайтом
connection.login()                        # по умолчанию анонимная регистрация
connection.cwd('pub/python/src')          # единовременная передача 1k в localfile

print 'Downloading...'
connection.retrbinary('RETR ' + tarname, localfile.write, 1024)
connection.quit()
localfile.close()
```

```
print 'Unpacking...'
os.system('gzip -d '  + tarname)          # декомпрессия
os.system('tar -xvf ' + tarname[:-3])     # отрезать .gz

print 'Building...'
os.chdir('Python-' + Version)             # компилировать сам Python
os.system('./configure')                  # предполагает make в стиле unix
os.system('make')
os.system('make test')
print 'Done: see Python-%s/python.' % Version
```

Большинство деталей протокола FTP инкапсулировано в импортируемом модулем Python `ftplib`. Данный сценарий использует самые простые интерфейсы `ftplib` (остальные мы увидим чуть позже), но они представляют модуль в целом:

```
connection = FTP('ftp.python.org')        # соединиться с ftp-сайтом
```

Чтобы открыть соединение с удаленным (или локальным) сервером FTP, нужно создать экземпляр объекта `ftplib.FTP`, передав ему имя (доменное или IP-адрес) машины, с которой нужно соединиться. Если при этом вызове не возбуждается исключительная ситуация, полученный объект FTP экспортирует методы, соответствующие обычным операциям FTP. В действительности сценарии Python действуют подобно типичным программам FTP-клиентов – нужно просто заменить обычные вводимые или выбираемые команды вызовами методов:

```
connection.login()                        # по умолчанию анонимная регистрация
connection.cwd('pub/python/src')          # одноразовая передача 1k в localfile
```

После соединения производится регистрация и переход в удаленный каталог, из которого нужно получить файл. Метод `login` позволяет передавать дополнительные необязательные аргументы, задающие имя пользователя и пароль. По умолчанию выполняется анонимная регистрация FTP:

```
connection.retrbinary('RETR ' + tarname, localfile.write, 1024)
connection.quit()
```

После перехода в целевой каталог вызывается метод `retrbinary` для загрузки с сервера целевого файла в двоичном режиме. Для завершения вызова `retrbinary` требуется некоторое время, поскольку должен быть загружен большой файл. Метод принимает три аргумента:

- Строка команды FTP, в данном случае строка `RETR` *имя_файла*, являющаяся стандартным форматом загрузки по FTP.
- Функция или метод, которым Python передает каждый блок загруженных байтов файла, в данном случае метод `write` вновь созданного и открытого локального файла.
- Размер этих блоков байтов, в данном случае каждый раз загружается 1024 байта, но если этот аргумент опущен, используется значение по умолчанию.

Так как этот сценарий создает локальный файл с именем `localfile`, таким же, как у загружаемого удаленного файла, и передает его метод `write` методу FTP-получения, содержимое удаленного файла автоматически окажется в локальном файле на стороне клиента после завершения загрузки. Между прочим, обратите внимание на открытие этого файла в двоичном режиме вывода «wb»: если этот сценарий выполняется под Windows, нужно избежать автоматического преобразования байтов `\n` в последовательности байтов `\r\n` (которое происходит в Windows при записи файлов, открытых в текстовом режиме «w»).

Наконец, вызывается метод FTP `quit`, чтобы разорвать соединение с сервером, и вручную закрывается локальный файл с помощью `close`, чтобы сделать файл законченным, прежде чем обрабатывать его командами оболочки, порождаемыми `os.system` (без вызова `close` части файла могут остаться в буферах):

```
connection.quit()
localfile.close()
```

Вот и все, что нужно сделать. Все детали FTP, сокетов и работы в сети скрыты в модуле интерфейса `ftplib`. Вот результаты работы этого сценария под Linux, из которых для краткости удалены несколько тысяч строк вывода:

```
[lutz@starship test]$ python getpython.py
Connecting...
Downloading...
Unpacking...
Python-1.5/
Python-1.5/Doc/
Python-1.5/Doc/ref/
Python-1.5/Doc/ref/.cvsignore
Python-1.5/Doc/ref/fixps.py
...
...удалена масса строк tar...
...
Python-1.5/Tools/webchecker/webchecker.py
Python-1.5/Tools/webchecker/websucker.py
Building...
creating cache ./config.cache
checking MACHDEP... linux2
checking CCC...
checking for --without-gcc... no
checking for gcc... gcc
...
...удалена масса строк компиляции...
...
Done: see Python-1.5/python.

[lutz@starship test]$ cd Python-1.5/
[lutz@starship Python-1.5]$ ./python
Python 1.5 (#1, Jul 12 2000, 12:35:52)  [GCC egcs-2.91.66 19990314/Li on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> print 'The Larch!'
The Larch!
```

Такой сценарий можно автоматически выполнять через регулярные интервалы времени (например, с помощью задания для *cron* в Unix), чтобы обновлять локальную установку Python. Внимание здесь нужно обратить на то, что этот обычный в остальных отношениях сценарий Python получает данные с произвольных удаленных сайтов FTP и машин. Если имеется ссылка Интернета, то *любая* информация, опубликованная на сервере FTP в Сети, может быть получена сценариями Python с помощью подобных интерфейсов.

Применение urllib для работы с файлами по FTP

В действительности FTP является лишь одним из способов передачи информации через Сеть, и в библиотеке Python есть более общие средства для выполнения загрузки, осуществляемой в предыдущем сценарии. Проще всего, видимо, работать с модулем Python `urllib`: получив строку с адресом в Интернете – URL, или унифицированный

указатель ресурса (Uniform Resource Locator) – этот модуль открывает соединение с указанным сервером и возвращает объект типа файла, который можно читать с помощью обычных вызовов методов объекта файла (например, `read`, `readlines`).

Такой интерфейс более высокого уровня может быть применен для загрузки всего, что имеет адрес в Сети – файлов, опубликованных на FTP-сайтах (при помощи URL, начинающихся с «ftp://»), веб-страниц и выходных данных сценариев, расположенных на удаленных серверах (при помощи URL, начинающихся с «http://»), локальных файлов (при помощи URL, начинающихся с «file://»), данных серверов Gopher и прочего. В частности, сценарий примера 11.2 делает то же, что пример 11.1, но использует для получения файла дистрибутива исходного кода общий модуль `urllib` вместо модуля конкретного протокола `ftplib`.

Пример 11.2. PP2E\Internet\Ftp\getpython-urllib.py

```
#!/usr/local/bin/python
########################################################################
# Сценарий Python для загрузки и компиляции исходного кода Python. Использует для получения
# файла более высокоуровневый urllib, а не ftplib. urllib поддерживает протоколы ftp, http,
# gopher и локальные файлы. urllib также позволяет загружать страницы html, изображения,
# текст и т. д.; см. также Python-анализаторы html/xml для веб-страниц, загружаемых urllib;
########################################################################
import os
import urllib                                # веб-средства, основанные на сокетах
Version = '1.5'                              # версия, которую нужно загрузить
tarname = 'python%s.tar.gz' % Version        # имя удаленного/локального файла

remoteaddr = 'ftp://ftp.python.org/pub/python/src/' + tarname
print 'Downloading', remoteaddr

# такой код тоже работает:
# urllib.urlretrieve(remoteaddr, tarname)

remotefile = urllib.urlopen(remoteaddr)    # возвращает объект типа файла для ввода
localfile  = open(tarname, 'wb')           # куда записать данные локально
localfile.write(remotefile.read())
localfile.close()
remotefile.close()

# остальное так же
execfile('buildPython.py')
```

Не волнуйтесь по поводу деталей использованной здесь строки URL: мы значительно подробнее поговорим об URL в следующей главе. Мы также вновь будем обращаться к `urllib` в этой и последующих главах для получения веб-страниц, форматирования сгенерированных строк URL и получения выходных данных удаленных сценариев в Сети.[1] Технически говоря, `urllib`, поддерживая целый ряд протоколов Интернета (HTTP, FTP, Gopher и локальные файлы), используется только для чтения удаленных объектов (не для записи или загрузки их на сервер), а получение данных обычно должно осуществляться в потоках, если блокировка составляет предмет для беспокойства. Однако базовый интерфейс, показанный в этом сценарии, прост. Вызов:

```
remotefile = urllib.urlopen(remoteaddr) # возвращает объект типа файл для ввода
```

[1] Дополнительные примеры загрузки посредством `urllib` можно найти в разделе этой главы, посвященном HTTP. Вообще говоря, такие инструменты, как `urllib.urlopen`, позволяют сценариям как загружать удаленные файлы, так и запускать программы, размещенные на удаленных серверах. В главе 12 «Сценарии, выполняемые на сервере» мы увидим также, что `urllib` содержит средства для форматирования (преобразования с помощью escape-последовательностей) строк URL для безопасности передачи.

соединяется с сервером, указанным в строке URL `remoteaddr` и возвращает объект типа файла, подключенный к потоку загрузки (сокету, использующему FTP). При вызове метода `read` вытаскивается содержимое файла, которое записывается в локальный файл клиента. Еще более простой интерфейс:

```
urllib.urlretrieve(remoteaddr, tarname)
```

также открывает локальный файл и записывает в него загружаемые байты, что в данном сценарии выполняется вручную. Такой интерфейс удобен, если нужно загрузить файл, но менее полезен, если требуется сразу же обрабатывать его данные.

В любом случае конечный результат один и тот же: требуемый файл сервера оказывается на машине клиента. Остальная часть сценария – распаковка и компиляция – такая же, как в первоначальной версии, поэтому она помещена в файл Python для многократного использования и выполняется с помощью встроенной функции `execfile` (напомним, что `execfile` выполняет файл так, как если бы его код был вставлен в то место, где находится `execfile`). Сценарий показан в примере 11.3.

Пример 11.3. PP2E\Internet\Ftp\buildPython.py

```
#!/usr/local/bin/python
################################################################
# Сценарий Python для сборки Python из его исходного кода.
# Выполнять в том каталоге, где находится исходный код Python.
################################################################

import os
Version = '1.5'                         # компилируемая версия
tarname = 'python%s.tar.gz' % Version   # имя удаленного/локального файла

print 'Unpacking...'
os.system('gzip -d '  + tarname)        # декомпрессия файла
os.system('tar -xvf ' + tarname[:-3])   # распаковка имени без '.gz'

print 'Building...'
os.chdir('Python-' + Version)           # компилировать сам Python
os.system('./configure')                # предполагает make в стиле unix
os.system('make')
os.system('make test')
print 'Done: see Python-%s/python.' % Version
```

Вывод в этом случае почти идентичен выводу примера 11.3, поэтому я покажу лишь небольшую его часть (сообщение `gzip` появится, если не удален tar-файл от прежнего прогона):

```
[lutz@starship test]$ python getpython-urllib.py
Downloading ftp://ftp.python.org/pub/python/src/python1.5.tar.gz
Unpacking...
gzip: python1.5.tar already exists; do you wish to overwrite (y or n)? y
...строки tar...
Building...
...строки компиляции...
Done: see Python-1.5/python.

[lutz@starship test]$ python buildPython.py
Unpacking...
...строки tar и компиляции...
```

В действительности хотя исходный сценарий целиком представляет собой код верхнего уровня, выполняемый непосредственно и решающий только одну задачу, есть две возможности повторного его использования: для получения файла и для компи-

ляции Python из исходного кода. Выделяя каждую часть в отдельный модуль, можно повторно использовать логику программы в других контекстах, что естественным образом приводит нас к теме следующего раздела.

Утилиты FTP get и put

Почти всегда, когда я рассказываю об интерфейсах `ftplib` на занятиях по Python, учащиеся интересуются, для чего программист должен задавать строку RETR в методе загрузки retrieval. Это правильный вопрос: строка RETR является названием команды загрузки в протоколе FTP, но `ftplib` должен *инкапсулировать* этот протокол. Как мы увидим несколько ниже, при загрузке на сервер также требуется задавать странную строку STOR. Это стереотипный код, принимаемый на веру, когда смотришь на него, но который напрашивается на этот вопрос. Вы всегда можете послать Гвидо по электронной почте предложение сделать заплатку для `ftplib`, но это не самый хороший ответ начинающим изучать Python.[1]

Лучше будет ответить так: нет закона, запрещающего расширять стандартные библиотечные модули собственными интерфейсами более высокого уровня – с помощью всего лишь нескольких строк многократно используемого кода можно заставить интерфейс FTP в Python выглядеть так, как вы захотите. Например, можно взять и написать вспомогательные модули, служащие оболочкой интерфейсов `ftplib`, скрывающей строку RETR. Если поместить эти модули в каталог, содержащийся в PYTHONPATH, они станут столь же доступными, как сам `ftplib`, и будут автоматически использоваться в любом сценарии Python, который может быть написан в будущем. Помимо устранения необходимости в строке RETR модуль-оболочка может использовать допущения, которые упрощают операции FTP до единственного вызова функции.

В частности, при наличии модуля, который инкапсулирует и упрощает `ftplib`, наш модуль для загрузки и компиляции Python можно сократить еще сильнее, что иллюстрирует сценарий из примера 11.4, в сущности, состоящий только из вызова функции и выполнения файла.

Пример 11.4. PP2E\Internet\Ftp\getpython-modular.py

```
#!/usr/local/bin/python
#########################################################################
# Сценарий Python для загрузки и компиляции исходного кода Python.
# Использует вспомогательный модуль getfile.py, инкапсулирующий этап ftp.
#########################################################################

import getfile
Version = '1.5'                         # версия, которую нужно загрузить
tarname = 'python%s.tar.gz' % Version   # имя удаленного/локального файла

# загрузить посредством вспомогательного модуля
getfile.getfile(tarname, 'ftp.python.org', 'pub/python/src')

# остальное без изменений
execfile('buildPython.py')
```

[1] Это один из моментов занятий, когда я обычно грожу написать на доске номер домашнего телефона Гвидо. Но в целом это пустое обещание, делаемое для смеха. Если вы действительно хотите принять участие в обсуждении проблем языка Python, то в Сети можно легко найти адрес электронной почты Гвидо и способы контакта с другими основными разработчиками Python. Как один из тех, кто получал анонимные домашние звонки, связанные с Python, я никогда не раскрываю телефонных номеров (а набрать 1-800-Hi-Guido забавно только первый раз).

Помимо того что количество строк в нем производит огромное впечатление на рекламных агентов, основное тело этого сценария разбито на отдельные файлы, которые можно повторно использовать в других ситуациях. Если когда-либо вновь потребуется загрузить файл, достаточно импортировать существующую функцию, а не заниматься редактированием путем удаления и вставки. Операции загрузки потребуется изменить только в одном файле, а не во всех местах, куда был скопирован стереотипный код; можно даже сделать так, чтобы `getfile.getfile` использовал `urllib` вместо `ftplib`, никак не затронув при этом его клиентов. Это хорошая конструкция.

Утилита для загрузки с сервера

И как же можно написать такую оболочку интерфейса FTP (спросит читатель понимающе)? При наличии библиотечного модуля `ftplib` создать оболочку для загрузки конкретного файла из конкретного каталога просто. Объекты FTP, осуществившие соединение, поддерживают два метода загрузки (download):

- Метод `retrbinary` загружает запрашиваемый файл в *двоичном* режиме, блоками посылая его байты указанной функции и не преобразуя символов конца строки. Обычно в качестве функции задается метод write объекта открытого локального файла, благодаря которому эти байты помещаются в локальный файл клиента.
- Метод `retrlines` загружает запрашиваемый файл в режиме *текста* ASCII, посылая заданной функции каждую строку текста с удаленными символами конца строки. Обычно заданная функция добавляет символ новой строки `\n` (преобразуемый соответственно машине клиента) и записывает строку в локальный файл.

Позднее мы встретимся с примером, использующим метод `retrlines`; вспомогательный модуль `getfile` в примере 11.5 осуществляет передачу в *двоичном* режиме всегда с помощью `retrbinary`. Это означает, что файлы загружаются в точности в том виде, в каком они находятся на сервере, байт в байт, сохраняя для текстовых файлов те символы конца строки, которые приняты на сервере. Если эти символы выглядят необычно в вашем текстовом редакторе, может потребоваться преобразовать их после загрузки – указания смотрите в средствах конвертирования, описываемых в главе 5 «Более крупные системные примеры, часть 2».

Пример 11.5. PP2E\Internet\Ftp\getfile.py

```
#!/usr/local/bin/python
################################################
# Получение произвольного файла с помощью ftp. Анонимный
# ftp, если не задать кортеж user=(имя, пароль).
# По умолчанию загружает музыкальную тему Monty Python.
################################################

from ftplib  import FTP          # средства ftp, основанные на сокетах
from os.path import exists       # проверка существования файла

file = 'sousa.au'                # координаты файла по умолчанию
site = 'ftp.python.org'          # музыкальная тема monty python
dir  = 'pub/python/misc'

def getfile(file=file, site=site, dir=dir, user=(), verbose=1, force=0):
    """

    загрузить файл по протоколу ftp с сайта/каталога
    используя анонимный или реальный вход, бинарный режим передачи
    """
    if exists(file) and not force:
        if verbose: print file, 'already fetched'
```

```
    else:
        if verbose: print 'Downloading', file
        local = open(file, 'wb')       # локальный файл с тем же именем
        try:
            remote = FTP(site)         # соединиться с ftp-сайтом
            apply(remote.login, user)  # для анонимного =() или (имя, пароль)
            remote.cwd(dir)
            remote.retrbinary('RETR ' + file, local.write, 1024)
            remote.quit()
        finally:
            local.close()                  # что бы ни было, закрыть файл
        if verbose: print 'Download done.' # исключительные ситуации обрабатывает вызвавший
if __name__ == '__main__': getfile()       # анонимная регистрация на python.org
```

Этот модуль в основном просто создает новую форму для кода FTP, использовавшегося выше для получения дистрибутива исходного кода Python, с целью сделать его более простым и многократно используемым. Так как это вызываемая функция, экспортируемая здесь `getfile.getfile` стремится быть возможно более надежной и широко используемой, но даже такая маленькая функция требует некоторых конструктивных решений. Вот несколько замечаний по использованию:

Режим FTP

Функция `getfile` в этом сценарии по умолчанию выполняется в режиме анонимного FTP, однако в качестве аргумента `user` можно передать кортеж, состоящий из строк имени пользователя и пароля, чтобы зарегистрироваться на удаленном сервере в неанонимном режиме. Для работы в режиме анонимного FTP не передавайте этот аргумент или передайте в нем пустой кортеж, `()`. Метод `login` объекта FTP позволяет задать два необязательных аргумента, обозначающие имя пользователя и пароль, а вызов `apply` в примере 11.5 отправляет ему тот кортеж аргумента, который был передан `user`.

Режимы обработки

Последние два аргумента (`verbose`, `force`), если они переданы, позволяют отключить сообщения о статусе, выводимые в поток `stdout` (возможно, нежелательные в контексте GUI), и вынудить выполнение загрузки даже в том случае, когда уже существует локальный файл (загрузка переписывает существующий файл).

Протокол обработки исключительных ситуаций

Предполагается, что исключительные ситуации обрабатывает вызвавший; данная функция заключает загрузку внутрь оператора `try/ finally`, чтобы гарантировать закрытие локального выходного файла, но позволяет исключительным ситуациям распространяться. Например, при использовании в GUI или вызове в потоке исключительные ситуации могут потребовать особой обработки, о которой этот файл ничего не знает.

Самотестирование

При самостоятельном выполнении этот сценарий загружает с целью самопроверки звуковой файл *sousa.au* с *http://www.python.org*, но этой функции также обычно передаются имена файлов, сайтов и каталогов для FTP.

Режим открытия файла

Этот сценарий сделан так, чтобы открывать локальный выходной файл в двоичном режиме «wb» для подавления преобразования символов конца строки в случае выполнения под Windows. Как известно из главы 2 «Системные инструменты», файлы с действительно двоичными данными могут содержать байты со значением `\n`, соответствующим символу перевода строки; открытие их в текстовом режиме

«w» приведет к автоматическому преобразованию этих байтов в последовательность \r\n при записи в Windows локально. Это связано только с переносимостью под Windows (режим «w» действует всюду). Опять же, читайте о средствах конвертирования перевода строки в главе 5.

Модель каталогов

Данная функция использует одно и то же имя для удаленного файла и локального файла, в котором должно быть сохранено загруженное содержимое. Поэтому ее следует выполнять в том каталоге, где должен оказаться загруженный файл. При необходимости переместиться в нужный каталог используется `os.chdir`. (Можно было бы сделать так, чтобы *filename* представлял имя локального файла, и убрать из него локальный каталог с помощью `os.path.split`, чтобы получить имя удаленного файла, либо принимать два различных аргумента имени файла – одно локальное и другое удаленное.)

Обратите также внимание, что несмотря на свое название, этот модуль весьма отличается от сценария *getfile.py*, изученного в конце материала по сокетам в предыдущей главе. Основанный на сокетах `getfile` реализовывал логику клиента и сервера для загрузки файла с сервера на машину клиента непосредственно через сокеты.

Этот новый `getfile` является инструментом только *клиентской стороны*. Для запроса файла с сервера вместо непосредственного использования сокетов в нем применяется более простой протокол FTP; все детали уровня сокетов скрыты в реализации FTP-протокола клиента модулем `ftplib`. Кроме того, сервер здесь является постоянно выполняемой на машине сервера программой, которая ждет запросов FTP на сокете и отвечает на них, используя выделенный порт FTP (номер 21). В итоге оказывается, что этому сценарию требуется, чтобы на машине, где находится нужный файл, работал сервер FTP, но наличие такого сервера весьма вероятно.

Утилита для загрузки файлов на сервер

Если уж мы занялись этим делом, напишем сценарий для загрузки (upload) по FTP одиночного файла на удаленную машину. Интерфейсы загрузки на сервер модуля FTP симметричны интерфейсам загрузки с сервера. Если есть осуществивший соединение объект FTP:

- Посредством его метода `storbinary` можно загружать на сервер байты из открытого объекта локального файла.
- Посредством его метода `storlines` можно загружать на сервер текст в режиме ASCII из открытого объекта локального файла.

В отличие от интерфейсов загрузки с сервера обоим этим методам передается целиком *объект* файла, а не метод этого объекта (или другая функция). С методом `storlines` мы еще встретимся позже. Вспомогательный модуль примера 11.6 использует `storbinary` таким образом, что файл, имя которого передается методу, всегда передается дословно – в двоичном режиме, без трансляции символов перевода строки соответственно соглашениям, принятым на целевой машине. Если этот сценарий посылает текстовый файл, он будет получен точно в том виде, в каком хранился на машине, с которой поступил, со всеми маркерами перевода строки у клиента.

Пример 11.6. PP2E\Internet\Ftp\putfile.py

```
#!/usr/local/bin/python
#####################################################
# Записать произвольный файл через ftp. Анонимный
# ftp, если не передать кортеж user=(имя, пароль).
#####################################################
```

```
import ftplib                    # инструменты ftp, основанные на сокетах

file = 'sousa.au'                # координаты файла по умолчанию
site = 'starship.python.net'     # музыкальная тема monty python
dir  = 'upload'

def putfile(file=file, site=site, dir=dir, user=(), verbose=1):
    """
    передать файл по протоколу ftp на сайт/каталог
    используя анонимный или реальный вход, бинарный режим передачи
    """
    if verbose: print 'Uploading', file
    local  = open(file, 'rb')               # локальный файл с тем же именем
    remote = ftplib.FTP(site)               # соединиться с ftp-сайтом
    apply(remote.login, user)               # анонимный или реальный login
    remote.cwd(dir)
    remote.storbinary('STOR ' + file, local, 1024)
    remote.quit()
    local.close()
    if verbose: print 'Upload done.'

if __name__ == '__main__':
    import sys, getpass
    pswd = getpass.getpass(site + ' pswd?')          # имя файла в командной строке
    putfile(file=sys.argv[1], user=('lutz', pswd))   # не анонимный login
```

Обратите внимание, что для переносимости локальный файл на этот раз открывается в двоичном режиме «rb», чтобы предотвратить автоматическое преобразование символа перевода строки в случае выполнения под Windows; если это двоичная информация, байты, значением которых окажется символ возврата каретки `\r`, не должны таинственно исчезнуть во время передачи.

Заметьте также, что для запроса пароля FTP в автономном режиме используется стандартный вызов Python `getpass.getpass`. Подобно встроенной функции `raw_input`, этот вызов выводит приглашение и считывает строку текста с консоли пользователя; в отличие от `raw_input`, `getpass` не отображает вводимые символы на экране (в действительности под Windows он использует низкоуровневый интерфейс прямого обращения к клавиатуре, с которым мы встречались в разделе главы 2, посвященном переадресации потоков). Это удобно для защиты паролей и подобных вещей от любопытных глаз.

Подобно утилите для получения файлов при самотестировании, этот сценарий по умолчанию отправляет локальный экземпляр звукового файла, но обычно ему передаются строки реальных имен файла, сайта и каталога. И так же, как в утилите получения файла, можно передать кортеж `(имя_пользователя, пароль)` в качестве аргумента `user` для работы в режиме неанонимного FTP (по умолчанию FTP – анонимный).

Воспроизведение музыкальной темы Monty Python

Встряхнитесь – пора несколько развлечься. Воспользуемся этими сценариями для передачи и воспроизведения звукового файла с музыкальной темой Monty Python, находящегося на сайте Python. Прежде всего, напишем модуль, который загружает и воспроизводит файл, как показано в примере 11.7.

Пример 11.7. PP2E\Internet\Ftp\sousa.py

```
#!/usr/local/bin/python
#########################################################################################
# Использование: % sousa.py
```

```
# Загрузить и воспроизвести музыкальную тему Monty Python. В настоящем виде
# может не работать на вашей системе: требует машину с доступом по ftp и использует
# аудиофильтры на Unix и плеере .au под Windows.
# Настройте playfile.py, как необходимо.
#########################################################################################

import os, sys
from PP2E.Internet.Ftp.getfile  import getfile
from PP2E.Internet.Ftp.playfile import playfile
sample = 'sousa.au'

getfile(sample)    # загрузить аудиофайл через ftp
playfile(sample)   # отправить его аудиоплееру
```

Этот сценарий будет работать на любой машине с Python, связью с Интернетом и зарегистрированным в системе аудиоплеером; он действует на моем переносном компьютере с Windows и соединением с Интернетом по коммутируемой линии (если бы это было возможно, я вставил бы гиперссылку на звуковой файл, чтобы показать, как он звучит):

```
C:\...\PP2E\Internet\Ftp>python sousa.py
Downloading sousa.au
Download done.

C:\...\PP2E\Internet\Ftp>python sousa.py
sousa.au already fetched
```

Модули `getfile` и `putfile` также могут использоваться для перемещения образца звукового файла. Оба они могут быть импортированы клиентами, желающими использовать их функции, или запущены как программы верхнего уровня, выполняющие самотестирование. Запустим их из командной строки и интерактивного приглашения и посмотрим, как они работают. При автономном выполнении в командной строке передаются параметры и используются настройки файла по умолчанию:

```
C:\...\PP2E\Internet\Ftp>python putfile.py sousa.au
starship.python.net pswd?
Uploading sousa.au
Upload done.
```

При импорте параметры явно передаются функциям:

```
C:\...\PP2E\Internet\Ftp>python
>>> from getfile import getfile
>>> getfile(file='sousa.au', site='starship.python.net', dir='upload',
...                          user=('lutz', '****'))
Downloading sousa.au
Download done.
>>> from playfile import playfile
>>> playfile('sousa.au')
```

В этой головоломке недостает одного элемента: необходимо написать модуль, который пытается воспроизводить файл переносимым образом (см. пример 11.8). К сожалению, это наименее простая задача, так как на каждой платформе действуют свои аудиоплееры. В Windows следующий модуль с помощью команды DOS *start* запускает ту программу, которая зарегистрирована для воспроизведения звуковых файлов (то, что происходит при двойном щелчке по значку файла в менеджере файлов); при работе на моем ноутбуке Sony под Windows 98 командная строка DOS:

```
C:\...\PP2E\Internet\Ftp>python playfile.py sousa.au
```

выводит панель проигрывателя, воспроизводящего этот файл. В Unix делается попытка передать звуковой файл программе командной строки для воспроизведения,

если таковая добавлена в таблицу `unixfilter`, – подправьте ее для своей системы (в некоторых системах Unix срабатывает копирование звуковых файлов в */dev/audio* с помощью *cat*). На других платформах требуются некоторые дополнительные действия. Для создания переносимых аудиоинтерфейсов в Python кое-что сделано, но здесь заметно проявляется специфичность платформ. Веб-броузеры обычно умеют воспроизводить звуковые файлы, поэтому переносимым решением может явиться передача имени файла из URL броузеру, обнаруживаемому с помощью сценария *LaunchBrowser.py*, с которым мы познакомились в главе 4 «Более крупные системные примеры, часть 1» (читайте там о деталях интерфейса).

Пример 11.8. PP2E\Internet\Ftp\playfile.py

```
#!/usr/local/bin/python
#########################################################################################
# Попытка воспроизвести произвольный звуковой файл. В данном виде может не работать
# на вашей системе; использует аудиофильтры в Unix и связь имен файлов в командной строке
# start под Windows (т. е. зависит от того, что зарегистрировано на машине для запуска
# файлов *.au - аудиоплеер или веб-броузер); сделайте соответствующие изменения. Можно было бы
# запускать здесь веб-броузер с помощью LaunchBrowser.py. См. также: Lib/audiodev.py.
#########################################################################################

import os, sys
sample = 'sousa.au'  # аудиофайл по умолчанию

unixhelpmsg = """
К сожалению, не удалось найти аудиофильтр для вашей системы!
Добавьте запись для своей системы в словарь "unixfilter"
в файле playfile.py или воспроизведите файл вручную.
"""

unixfilter = {'sunos5':  '/usr/bin/audioplay',
              'linux2':  '<unknown>',
              'sunos4':  '/usr/demo/SOUND/play'}

def playfile(sample=sample):
    """
    воспроизведение аудиофайла: используется ассоциированность
    имен файлов в WIndows и командные строки фильтров в других системах
    """
    if sys.platform[:3] == 'win':
        os.system('start ' + sample)   # запускает аудиоплеер
    else:
        if not (unixfilter.has_key(sys.platform) and
                os.path.exists(unixfilter[sys.platform])):
            print unixhelpmsg
        else:
            theme = open(sample, 'r')
            audio = os.popen(unixfilter[sys.platform], 'w')     # запустить утилиту
                                                                #       оболочки
            audio.write(theme.read())                           # отправить файл на ее stdin

if __name__ == '__main__': playfile()
```

Добавление интерфейсов пользователя

Если вы читали предыдущую главу, то должны вспомнить, что она завершалась кратким обзором сценариев, добавляющих интерфейс пользователя к основанному на сокетах сценарию `getfile` – он передавал файлы через специфический диалог *сокетов*, а не FTP. В конце той презентации было отмечено, что FTP предоставляет гораздо бо-

лее полезный способ перемещения файлов, потому что серверы FTP широко распространены в Сети. В иллюстративных целях пример 11.9 показывает простое видоизменение пользовательского интерфейса из предыдущей главы, реализованное как новый подкласс общего построителя форм из предыдущей главы.

Пример 11.9. P2E\Internet\Ftp\getfilegui.py

```
############################################################################################
# Запуск функции ftp getfile с многократно используемым классом form gui; с помощью os.chdir
# переходит в целевой локальный каталог (getfile в настоящее время предполагает,
# что у имени файла нет префикса пути локального каталога); выполняет getfile.getfile в потоке,
# что позволяет выполнять их несколько одновременно и избегать блокировок gui
# во время загрузки; отличается от основанного на сокетах getfilegui, но повторно
# использует Form; в данном виде поддерживает ftp с заданием пользователя и анонимный;
# предостережение: поле пароля не выводится в виде звездочек, ошибки выводятся на консоль,
# а не в gui (потоки не могут обращаться к gui под Windows), безопасность потоков обеспечена
# не на все 100% (есть небольшая задержка между os.chdir и открытием локального выходного файла
# в getfile) и можно было бы выводить всплывающее окно "сохранить как" для выбора локального
# каталога и содержимое удаленного каталога для выбора загружаемого файла;
############################################################################################

from Tkinter import Tk, mainloop
from tkMessageBox import showinfo
import getfile, os, sys, thread              # здесь getfile с ftp, а не сокетами
from PP2E.Internet.Sockets.form import Form  # использовать инструмент form из каталога socket
class FtpForm(Form):
    def __init__(self):
        root = Tk()
        root.title(self.title)
        labels = ['Server Name', 'Remote Dir', 'File Name',
                  'Local Dir',   'User Name?', 'Password?']
        Form.__init__(self, labels, root)
        self.mutex = thread.allocate_lock()
        self.threads = 0
    def transfer(self, filename, servername, remotedir, userinfo):
        try:
            self.do_transfer(filename, servername, remotedir, userinfo)
            print '%s of "%s" successful'  % (self.mode, filename)
        except:
            print '%s of "%s" has failed:' % (self.mode, filename),
            print sys.exc_info()[0], sys.exc_info()[1]
        self.mutex.acquire()
        self.threads = self.threads - 1
        self.mutex.release()
    def onSubmit(self):
        Form.onSubmit(self)
        localdir   = self.content['Local Dir'].get()
        remotedir  = self.content['Remote Dir'].get()
        servername = self.content['Server Name'].get()
        filename   = self.content['File Name'].get()
        username   = self.content['User Name?'].get()
        password   = self.content['Password?'].get()
        userinfo   = ()
        if username and password:
            userinfo = (username, password)
        if localdir:
            os.chdir(localdir)
        self.mutex.acquire()
```

```
            self.threads = self.threads + 1
            self.mutex.release()
            ftpargs = (filename, servername, remotedir, userinfo)
            thread.start_new_thread(self.transfer, ftpargs)
            showinfo(self.title, '%s of "%s" started' % (self.mode, filename))
        def onCancel(self):
            if self.threads == 0:
                Tk().quit()
            else:
                showinfo(self.title,
                         'Cannot exit: %d threads running' % self.threads)

    class FtpGetfileForm(FtpForm):
        title = 'FtpGetfileGui'
        mode  = 'Download'
        def do_transfer(self, filename, servername, remotedir, userinfo):
            getfile.getfile(filename, servername, remotedir, userinfo, 0, 1)

    if __name__ == '__main__':
        FtpGetfileForm()
        mainloop()
```

Если перелистать страницы в конец предыдущей главы, то можно обнаружить, что эта версия по структуре аналогична той, которая приведена там; в действительности они одинаково названы (и различны только потому, что лежат в разных каталогах). Однако в последнем примере класс умеет пользоваться основанным на FTP модулем `getfile`, приведенным в начале данной главы, вместо основанного на сокетах модуля `getfile`, с которым мы познакомились в предыдущей главе. При выполнении данной версии также создается больше полей для ввода, как видно из рис. 11.1.

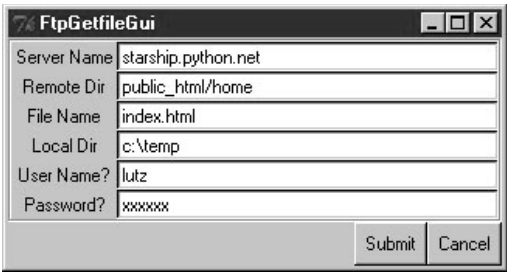


Рис. 11.1. Входная форма FTP getfile

Обратите внимание на ввод полного пути локального каталога. Если его не указать, сценарий предполагает текущий рабочий каталог, который изменяется после каждой загрузки с сервера и может зависеть от того, где запускается GUI (то есть текущий каталог будет иным, когда сценарий запускается из программы `PyDemos` в вершине дерева примеров). При щелчке по кнопке Submit в этом GUI (или нажатии клавиши <Enter>) данный сценарий просто передает значения полей ввода формы как аргументы функции FTP `getfile.getfile`, показанной выше в этом разделе. Кроме того, он показывает окно, сообщающее о начале загрузки (рис. 11.2).

В соответствии с данным кодом последующие сообщения о состоянии загрузки выводятся в окне консоли; вот сообщения при успешной загрузке, а также при неудавшейся из-за ошибки при вводе пароля (в действительности он, конечно, не «xxxxxx»):

```
User Name?      =>      lutz
Server Name     =>      starship.python.net
```

Рис. 11.2. Информационное окно FTP getfile

```
Local Dir       =>      c:\temp
Password?       =>      xxxxxx
File Name       =>      index.html
Remote Dir      =>      public_html/home
Download of "index.html" successful

User Name?      =>      lutz
Server Name     =>      starship.python.net
Local Dir       =>      c:\temp
Password?       =>      xxxxxx
File Name       =>      index.html
Remote Dir      =>      public_html/home
Download of "index.html" has failed: ftplib.error_perm 530 Login incorrect.
```

При наличии имени пользователя и пароля загрузчик производит регистрацию под определенной учетной записью. Для анонимного FTP нужно оставить поля имени пользователя и пароля пустыми. Давайте выполним анонимное FTP-соединение и получим дистрибутив исходного кода Python; на рис. 11.3 представлена заполненная форма.

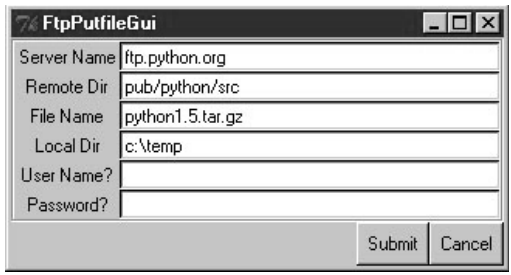


Рис. 11.3. Входная форма FTP getfile, анонимный FTP

При нажатии на этой форме Submit, как и прежде, начинается загрузка в фоновом режиме; начало загрузки подтверждается всплывающим окном, показанным на рис. 11.4.

Рис. 11.4. Информационное окно FTP getfile

Теперь, чтобы проиллюстрировать многопоточные возможности данного GUI, начнем еще одну загрузку, пока идет данная. GUI остается активным во время загрузки, по-

этому просто изменим значения полей ввода и еще раз нажмем кнопку Submit, как показано на рис. 11.5.

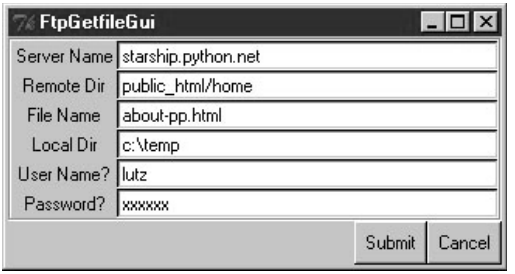


Рис. 11.5. Входная форма FTP getfile, второй поток

Эта вторая загрузка начинается параллельно с той, которая производится с *ftp.python.org*, так как каждая загрузка выполняется в потоке и одновременно могут быть активными несколько соединений с Интернетом. В действительности сам GUI остается активным во время загрузки только потому, что загрузка происходит в самостоятельном потоке; если бы это было не так, даже перерисовка экрана не происходила бы до завершения загрузки.

Мы обсуждали потоки в главе 3 «Системные средства параллельного выполнения», но данный сценарий иллюстрирует некоторые практические аспекты использования потоков:

- Эта программа избегает каких-либо действий с GUI в потоках загрузки. В Windows, по меньшей мере в настоящих версиях, работать с GUI может только тот поток, который его создает (правило, обусловленное только Windows, но не Python или Tkinter).
- Чтобы избежать закрытия порожденных потоков загрузки на некоторых платформах, GUI не должен завершаться, пока происходит какая-либо загрузка. Он следит за количеством потоков, в которых производится загрузка, и выводит окно, показанное на рис. 11.6, если попытаться закрыть GUI нажатием кнопки Cancel во время загрузки.

Рис. 11.6. Окно, сообщающее о занятости FTP getfile

Способ обойти правило, запрещающее трогать GUI из потоков, будет показан в примере `PyMailGui` ближе к концу этой главы. Однако для обеспечения переносимости действительно нельзя закрывать GUI, пока счетчик активных потоков не упадет до нуля. Вот какого рода вывод появляется в окне консоли для этих двух загрузок:

```
C:\...\PP2E\Internet\Ftp>python getfilegui.py
User Name?      =>
Server Name     =>      ftp.python.org
Local Dir       =>      c:\temp
Password?       =>
```

```
File Name       =>      python1.5.tar.gz
Remote Dir      =>      pub/python/src

User Name?      =>      lutz
Server Name     =>      starship.python.net
Local Dir       =>      c:\temp
Password?       =>      xxxxxx
File Name       =>      about-pp.html
Remote Dir      =>      public_html/home
Download of "about-pp.html" successful
Download of "python1.5.tar.gz" successful
```

Все это, конечно, не намного полезнее, чем утилита, основанная на командной строке, но данный код Python можно легко модифицировать, чтобы создать GUI, который можно будет считать простым наброском интерфейса пользователя FTP. Кроме того, поскольку этот GUI выполняет загрузку в потоках Python, из него можно одновременно выполнять несколько загрузок, не запуская новых утилит клиентов FTP.

Пока мы озабочены GUI, добавим также простой интерфейс и к утилите `putfile`. Сценарий примера 11.10 создает диалог для запуска отправки на сервер в потоках. Он почти не отличается от только что написанного GUI для `getfile`, поэтому ничего нового о нем сказать нельзя. В действительности, поскольку операции приема и отправки столь схожи с точки зрения интерфейса, значительная часть логики формы для получения была умышленно выделена в один родовой класс (`FtpForm`), благодаря чему изменения требуется производить только в одном месте. Таким образом, GUI для отправки здесь, по большей части, повторно использует GUI получения, с измененными метками и методом передачи. Он находится в отдельном файле, что облегчает его запуск как самостоятельной программы.

Пример 11.10. PP2E\Internet\Ftp\putfilegui.py

```
##########################################################################################################
# Запуск функции ftp putfile с многократно используемым классом form gui;
# см. примечания в getfilegui: применимо большинство тех же предупреждений; формы для get
# и put выделены в единый класс, чтобы производить изменения лишь в одном месте;
##########################################################################################################

from Tkinter import mainloop
import putfile, getfilegui

class FtpPutfileForm(getfilegui.FtpForm):
    title = 'FtpPutfileGui'
    mode  = 'Upload'
    def do_transfer(self, filename, servername, remotedir, userinfo):
        putfile.putfile(filename, servername, remotedir, userinfo, 0)

if __name__ == '__main__':
    FtpPutfileForm()
    mainloop()
```

Выполнение этого сценария весьма похоже на запуск GUI получения файлов, поскольку выполняемый код почти целиком тот же самый. Отправим парочку файлов с клиентской машины на сервер *starship*; на рис.11.7 показано состояние GUI во время отправки одного из файлов.

А вот вывод в окне консоли при параллельной отправке двух файлов; для отправки здесь снова используются потоки, поэтому, если начать новую отправку прежде, чем закончится текущая, они перекроются по времени:

```
User Name?      =>      lutz
Server Name     =>      starship.python.net
```

Рис. 11.7. Входная форма FTP putfile

```
Local Dir       =>      c:\stuff\website\public_html
Password?       =>      xxxxxx
File Name       =>      about-pp2e.html
Remote Dir      =>      public_html

User Name?      =>      lutz
Server Name     =>      starship.python.net
Local Dir       =>      c:\stuff\website\public_html
Password?       =>      xxxxxx
File Name       =>      about-ppr2e.html
Remote Dir      =>      public_html
Upload of "about-pp2e.html" successful
Upload of "about-ppr2e.html" successful
```

Наконец, можно увязать оба GUI с единым запускающим сценарием, который умеет открывать интерфейсы для получения и отправки независимо от того, в каком каталоге мы находимся при запуске сценария, и от платформы, на которой он выполняется. Этот процесс показан в примере 11.11.

Пример 11.11. PP2E\Internet\Ftp\PyFtpGui.pyw

```
#########################################################################################
# Породить gui для получения и отправки по ftp независимо от каталога, в котором
# находится сценарий; сценарий не обязательно находится в os.getcwd; можно также зашить
# путь от $PP2EHOME или guessLocation; понадобилось бы всплывающее окно DOS
# для сообщений о статусе: from PP2E.launchmodes import PortableLauncher
# PortableLauncher('getfilegui', '%s/getfilegui.py' % mydir)()
#########################################################################################

import os, sys
from PP2E.Launcher import findFirst
mydir = os.path.split(findFirst(os.curdir, 'PyFtpGui.pyw'))[0]

if sys.platform[:3] == 'win':
    os.system('start %s/getfilegui.py' % mydir)
    os.system('start %s/putfilegui.py' % mydir)
else:
    os.system('python %s/getfilegui.py &' % mydir)
    os.system('python %s/putfilegui.py &' % mydir)
```

При запуске этого сценария оба GUI – `get` и `put` – появляются как отдельные независимо выполняемые программы; альтернативой может быть прикрепление обеих форм к единому интерфейсу. Конечно, можно создать значительно более замысловатые интерфейсы. Например, можно воспользоваться всплывающими диалогами для выбора локальных файлов и отобразить графические элементы, сообщающие о состоянии текущего примема или отправки. Можно даже отобразить доступные на удаленном сер-

вере файлы в окне списка, запросив содержимое удаленного каталога через соединение FTP. Однако чтобы научиться добавлять такие функции, нужно перейти к следующему разделу.

Загрузка сайтов (создание зеркальных копий)

В былые времена мне не требовалось ничего, кроме telnet. Мой сайт размещался у провайдера Интернета (ISP), который предоставлял всем своим клиентам общий и бесплатный доступ через telnet. Тогда все было просто. Все файлы моего сайта находились в одном месте – в моем личном каталоге на сервере ISP. Чтобы изменить веб-страницу, я соединялся с машиной провайдера в сеансе telnet и редактировал веб-страницы в онлайновом режиме. Более того, поскольку сеанс telnet можно запустить почти с любой машины, подключенной к Интернету, я имел возможность корректировать свои страницы отовсюду – со своего PC, с машин, к которым у меня был доступ во время преподавательской работы, с доисторических машин, на которых я развлекался, устав от работы, и т. д. Жизнь была прекрасна.

Но времена меняются. Из-за бреши в системе защиты мой провайдер принял решение полностью запретить доступ по telnet для всех своих клиентов (кроме тех, конечно, которые согласились на значительную плату за то, чтобы сохранить его). По-видимому, предполагалось, что мы, прежде всего, даже не должны были знать о telnet. В обмен провайдер определил, что все склонные к использованию telnet пользователи должны хранить свои веб-страницы локально и после каждого изменения загружать их на сервер по FTP.

Это, конечно, далеко не так удобно, как редактировать файлы, хранящиеся в одном месте, с любой точки света, и вызвало массу жалоб и отказов от обслуживания у технически грамотных подписчиков. К несчастью, технически грамотные составляют незначительное в финансовом отношении подмножество; более важно, что к тому времени адрес моей веб-страницы был опубликован во многих книгах, продаваемых по всему свету, поэтому смена провайдера была бы не менее болезненной, чем смена процедуры обновления.

Успокоившись, я сообразил, что в этой ситуации мне может помочь Python: написав сценарии Python, автоматизирующие задачи отправки и приема файлов, связанные с сопровождением моего сайта, я мог хотя бы частично восстановить утраченные мобильность и простоту пользования. Так как сценарии Python для FTP могут работать на любой машине, поддерживающей сокеты, я могу выполнять их на своем PC и почти на всякой другой машине с установленным Python. Более того, те же сценарии, с помощью которых файлы страниц перемещаются на мой PC и с него, можно использовать для копирования (создания «зеркала») моего сайта на другой веб-сервер с целью создания резервной копии на случай, если возникнет остановка в работе моего провайдера (поверьте мне, и такое случается).

Следствием описанных несчастий явились следующие два сценария. Первый из них, *mirrorflat.py*, автоматически скачивает (то есть копирует) с помощью FTP все файлы из каталога на удаленном сайте в каталог на локальной машине. В настоящее время я храню основную копию файлов моего сайта на своем PC, но в действительности использую этот сценарий двумя способами:

- Чтобы загрузить мой сайт на клиентскую машину, где я хочу заняться редактированием, я получаю содержимое моего каталога *public_html* своей учетной записи на машине моего Интернет-провайдера.
- Чтобы сделать зеркальную копию моего сайта не сервере *starship.python.net*, где у меня есть учетная запись, я периодически выполняю этот сценарий в сеансе telnet

на машине *starship* (когда я это писал, *starship* все еще придерживался радикальной мысли, что пользователи достаточно подготовлены, чтобы запускать telnet).

Вообще говоря, этот сценарий (показанный в примере 11.12) скачает полный каталог файлов на любую машину с Python и сокетами с любой машины, на которой работает сервер FTP.

Пример 11.12. PP2E\Internet\Ftp\mirrorflat.py

```
#!/bin/env python
#########################################################################################
# С помощью ftp скопировать (загрузить) все файлы из каталога удаленного сайта в каталог
# локальной машины; например можно периодически делать зеркальную копию плоского ftp-сайта;
#########################################################################################
import os, sys, ftplib
from getpass import getpass

remotesite = 'home.rmi.net'
remotedir  = 'public_html'
remoteuser = 'lutz'
remotepass = getpass('Please enter password for %s: ' % remotesite)
localdir   = (len(sys.argv) > 1 and sys.argv[1]) or '.'
if sys.platform[:3] == 'win': raw_input() # очистить поток
cleanall   = raw_input('Clean local directory first? ')[:1] in ['y', 'Y']

print 'connecting...'
connection = ftplib.FTP(remotesite)            # соединиться с ftp-сайтом
connection.login(remoteuser, remotepass)       # зарегистрироваться как пользователь/пароль
connection.cwd(remotedir)                      # перейти в копируемый каталог

if cleanall:
    for localname in os.listdir(localdir):     # сначала попробовать удалить
        try:                                   # все старые локальные файлы
            print 'deleting local', localname
            os.remove(os.path.join(localdir, localname))
        except:
            print 'cannot delete local', localname

count = 0                                  # загрузить все удаленные файлы
remotefiles = connection.nlst()            # nlst() возвращает список файлов
                                           # dir() возвращает все подробности
for remotename in remotefiles:
    localname = os.path.join(localdir, remotename)
    print 'copying', remotename, 'to', localname
    if remotename[-4:] == 'html' or remotename[-3:] == 'txt':
        # передача в режиме ascii
        localfile = open(localname, 'w')
        callback  = lambda line, file=localfile: file.write(line + '\n')
        connection.retrlines('RETR ' + remotename, callback)
    else:
        # передача в двоичном режиме
        localfile = open(localname, 'wb')
        connection.retrbinary('RETR ' + remotename, localfile.write)
    localfile.close()
    count = count+1

connection.quit()
print 'Done:', count, 'files downloaded.'
```

В этом примере не так много нового в сравнении с другими примерами FTP, которые мы видели. Открываем соединение с удаленным сервером FTP, регистрируемся с не-

обходимыми именем пользователя и паролем (в этом сценарии не используется анонимный FTP) и переходим в требуемый удаленный каталог. Новыми здесь, однако, являются циклы, совершающие обход всех файлов в локальном и удаленном каталогах, загрузка в текстовом режиме и удаление файлов:

Удаление всех локальных файлов

У этого сценария есть параметр `cleanall`, который можно установить в ответ на интерактивный запрос. Если он включен, сценарий перед загрузкой удаляет все файлы из локального каталога, чтобы гарантировать отсутствие файлов, которых нет на сервере (они могут сохраниться от предыдущей загрузки). Чтобы удалить локальные файлы, сценарий вызывает `os.listdir` и получает список имен файлов в каталоге, затем удаляет каждый из них с помощью `os.remove`; если вы забыли, как работают эти вызовы, смотрите детали выше в главе 2 (или руководстве по библиотеке Python).

Обратите внимание на то, как `os.path.join` объединяет путь к каталогу с именем файла в соответствии с соглашениями, принятыми на платформе хоста: `os.listdir` возвращает имена файлов без путей, а этот сценарий не обязательно выполняется в локальном каталоге, куда будут помещены загружаемые файлы. Локальным каталогом по умолчанию станет текущий каталог («.»), но его можно изменить с помощью аргумента командной строки, передаваемого сценарию.

Загрузка всех удаленных файлов

Чтобы вытащить из удаленного каталога все файлы, сначала нужно получить список их имен. Объект FTP обладает методом `nlst`, служащим удаленным эквивалентом `os.listdir`: `nlst` возвращает список строк – имен всех файлов в текущем удаленном каталоге. Получив такой список, мы просто проходим его в цикле и выполняем FTP-команды получения файлов поочередно для каждого имени файла (подробнее об этом чуть ниже).

Метод `nlst` в некоторой мере похож на запрос списка содержимого каталога командой `ls` в обычных интерактивных программах FTP, но Python автоматически выделяет из текста листинга список имен файлов. Методу можно передать имя удаленного каталога, содержимое которого нужно показать; по умолчанию он обрабатывает текущий каталог сервера. Родственный метод FTP, `dir`, возвращает список строк, порожденных командой FTP *LIST*; результат похож на ввод команды *dir* в сеансе FTP, а строки, в отличие от `nlst`, содержат полные сведения о файлах. Если нужно больше узнать обо всех удаленных файлах, следует вызвать метод `dir` и проанализировать его результаты.

Получение текстовых файлов

Чтобы символы перевода строки соответствовали машинам, на которых расположены мои веб-файлы, этот сценарий различает двоичные и текстовые файлы. Для этого в нем применена простая эвристика: имена файлов, оканчивающиеся на *.html* или *.txt*, предполагают текстовые данные ASCII (веб-страницы HTML и простые текстовые файлы), а все остальные файлы предполагаются двоичными (например, графические файлы GIF и JPEG, звуковые файлы, tar-архивы). Это простое правило действует не с любым сайтом, но на моем это проходит.

Двоичные файлы загружаются с помощью метода `retrbinary`, с которым мы познакомились ранее, и локального режима открытия «wb», чтобы подавить преобразование байтов перевода строки (этот сценарий может выполняться в Windows или на Unix-подобных платформах). При этом здесь не используется третий аргумент размера блока – по умолчанию он принимает разумное значение 8 Кбайт.

Для текстовых файлов ASCII этот сценарий использует другой метод – `retrlines`, передавая ему функцию, которая должна вызываться для каждой загруженной строки текстового файла. Функция-обработчик строки текста обычно просто записывает строку в локальный файл. Но обратите внимание, что функция-обработчик, создаваемая здесь посредством `lambda`, добавляет также символ перевода строки `\n` в конец строки, которая ей передана. Метод Python `retrlines` удаляет из строк все символы перевода строки, чтобы обойти различия между платформами. В результате добавления `\n` сценарий обеспечивает добавление правильной последовательности символов маркера перевода строки для той платформы, на которой выполняется сценарий (`\n` или `\r\n`). Конечно, для того чтобы такое автоматическое отображение `\n` действовало в сценарии, необходимо также открытие выходных текстовых файлов в текстовом режиме «w», а не в режиме «wb» – при записи данных в файл под Windows происходит отображение `\n` в `\r\n`.

Все это проще на практике, чем на словах. Вот команда, которую я использую для загрузки всего моего сайта с сервера Интернет-провайдера на переносной компьютер с Windows 98 за один шаг:

```
C:\Stuff\Website\public_html>python %X%\internet\ftp\mirrorflat.py
Please enter password for home.rmi.net:
Clean local directory first?
connecting...
copying UPDATES to .\UPDATES
copying PythonPowered.gif to .\PythonPowered.gif
copying Pywin.gif to .\Pywin.gif
copying PythonPoweredAnim.gif to .\PythonPoweredAnim.gif
copying PythonPoweredSmall.gif to .\PythonPoweredSmall.gif
copying about-hopl.html to .\about-hopl.html
copying about-lp.html to .\about-lp.html
...
...здесь удалены строки...
...
copying training.html to .\training.html
copying trainingCD.GIF to .\trainingCD.GIF
copying uk-1.jpg to .\uk-1.jpg
copying uk-2.jpg to .\uk-2.jpg
copying uk-3.jpg to .\uk-3.jpg
copying whatsnew.html to .\whatsnew.html
copying whatsold.html to .\whatsold.html
copying xlate-lp.html to .\xlate-lp.html
copying uploadflat.py to .\uploadflat.py
copying ora-lp-france.gif to .\ora-lp-france.gif
Done: 130 files downloaded.
```

Для завершения этого может потребоваться изрядное время (что обусловлено ограничениями скорости передачи в сети), но такая процедура значительно точнее и проще, чем загрузка файлов вручную. Этот сценарий проходит весь список удаленных файлов, возвращенный методом `nlst`, и поочередно загружает каждый по протоколу FTP (то есть через сокеты). Для имен, очевидно указывающих на текстовые данные, используется текстовый режим передачи, а для остальных имен – двоичный режим.

Для такого сценария я проверяю, чтобы начальные присвоения в нем отражали участвующие в обмене машины, а затем запускаю его из локального каталога, в который хочу поместить копию сайта. Поскольку каталог для загрузки обычно отличен от того каталога, где находится сценарий, необходимо указать Python полный путь к файлу сценария (на моей машине `%X%` является переменной оболочки, содержащей путь верхнего уровня к примерам из книги). При выполнении на сервере *starship* в

окне сеанса telnet пути для выполнения и сценария отличаются, но сценарий работает одинаково.

Если вы решите удалить локальные файлы из каталога загрузки, то можете получить на экране серию сообщений «deleting local...» и только потом строки «copying...»:

```
...
deleting local uploadflat.py
deleting local whatsnew.html
deleting local whatsold.html
deleting local xlate-lp.html
deleting local old-book.html
deleting local about-pp2e.html
deleting local about-ppr2e.html
deleting local old-book2.html
deleting local mirrorflat.py
...
copying about-pp-japan.html to ./about-pp-japan.html
copying about-pp.html to ./about-pp.html
copying about-ppr-germany.html to ./about-ppr-germany.html
copying about-ppr-japan.html to ./about-ppr-japan.html
copying about-ppr-toc.html to ./about-ppr-toc.html
...
```

Кстати, если пароль удаленного сайта введен неудачно, возбуждается исключительная ситуация Python; мне иногда приходится запускать сценарий заново (и печатать медленнее):

```
C:\Stuff\Website\public_html>python %X%\internet\ftp\mirrorflat.py
Please enter password for home.rmi.net:
Clean local directory first?
connecting...
Traceback (innermost last):
  File "C:\PP2ndEd\examples\PP2E\internet\ftp\mirrorflat.py", line 20, in ?
    connection.login(remoteuser, remotepass)     # регистрироваться как пользователь.пароль
  File "C:\Program Files\Python\Lib\ftplib.py", line 316, in login
    if resp[0] == '3': resp = self.sendcmd('PASS ' + passwd)
  File "C:\Program Files\Python\Lib\ftplib.py", line 228, in sendcmd
    return self.getresp()
  File "C:\Program Files\Python\Lib\ftplib.py", line 201, in getresp
    raise error_perm, resp
ftplib.error_perm: 530 Login incorrect.
```

Следует отметить, что этот сценарий частично настраивается присваиваниями, производимыми в начале файла. Кроме того, опции пароля и удаления файлов указываются при интерактивном вводе, а аргумент командной строки разрешен один – с именем локального каталога для загрузки файлов (по умолчанию используется «.», то есть каталог, в котором выполняется сценарий). Можно было бы использовать аргументы командной строки для настройки других параметров и опций загрузки, но благодаря простоте Python и отсутствию шагов компиляции/сборки изменять настройки в тексте сценариев Python обычно не труднее, чем вводить слова в командной строке.

Замечание по вводу в Windows: если внимательно изучить предшествующий код, то можно заметить, что в Windows осуществляется дополнительный вызов `raw_input` после вызова ввода пароля `getpass` и перед вводом значения опции `cleanall`. Цель этого вызова – справиться с тем, что, по-видимому, является ошибкой в Python 1.5.2 для Windows.

Странно, но Windows-версия иногда не синхронизирует входной и выходной потоки командной строки должным образом. Здесь это, видимо, вызвано ошибкой или ограничениями getpass. Так как getpass использует интерфейс клавиатуры низкого уровня msvcrt, встречавшийся нам в главе 2, он плохо сочетается с буферизацией потока stdin, используемой raw_input, и в процессе портит входной поток. Дополнительный raw_input очищает входной поток (sys.stdin.flush не помогает).
В действительности без этого лишнего raw_input для Windows сценарий выводит приглашение для ввода опции cleanall, но не останавливается, чтобы позволить его ввести! В результате cleanall отключается вовсе. Чтобы получить различные входные и выходные строки и исправить поведение raw_input, некоторые сценарии в этой книге выполняют дополнительные команды print или вызовы raw_input, синхронизируя потоки перед тем, как продолжить взаимодействие с пользователем. Могут быть и другие способы обхода, а в следующих версиях это может быть исправлено: попробуйте выполнить сценарий без лишнего raw_input, чтобы проверить, не исправлена ли ошибка в вашей версии Python.

Загрузка сайтов на сервер

Загрузка на сервер целого каталога симметрична загрузке с сервера: в основном требуется поменять местами локальные и удаленные машины и операции в только что рассмотренной программе. Сценарий примера 11.13 с помощью FTP копирует все файлы каталога на локальной машине, на которой он выполняется, в каталог на удаленной машине.

Я действительно пользуюсь и этим сценарием, чаще всего для загрузки одним махом всех файлов, которые поддерживаю на своем переносном PC, на свою учетную запись у Интернет-провайдера. Иногда я также с его помощью копирую свой сайт с PC на зеркальный сервер *starship* или с зеркального сервера обратно к провайдеру. Этот сценарий выполняется на любом компьютере, где есть Python и сокеты, благодаря чему он может копировать каталог с любой машины, находящейся в Сети, на любую машину, где работает FTP-сервер. Нужно лишь поменять соответствующим образом начальные настройки в этом модуле, чтобы осуществить требуемую пересылку.

Пример 11.13. PP2E\Internet\Ftp\uploadflat.py

```
#!/bin/env python
#########################################################################
# С помощью ftp загрузить все файлы из локального каталога на удаленный сайт/каталог;
# например, запускать для копирования файлов веб/ftp-сайта с PC на ISP; предполагается
# загрузка плоского каталога: uploadall.py выполняет копирование вложенных каталогов.
# для действий с ISP я изменяю настройки на 'home.rmi.net' и 'public_html'.
#########################################################################

import os, sys, ftplib, getpass

remotesite = 'starship.python.net'              # загрузка на сайт starship
remotedir  = 'public_html/home'                 # с лэптопа под win или другого
remoteuser = 'lutz'
remotepass = getpass.getpass('Please enter password for %s: ' % remotesite)
localdir   = (len(sys.argv) > 1 and sys.argv[1]) or '.'
if sys.platform[:3] == 'win': raw_input()       # очистить поток
```

```
cleanall   = raw_input('Clean remote directory first? ')[:1] in ['y', 'Y']

print 'connecting...'
connection = ftplib.FTP(remotesite)              # соединиться с сайтом ftp
connection.login(remoteuser, remotepass)         # зарегистрироваться как пользователь/пароль
connection.cwd(remotedir)                        # переход в нужный каталог

if cleanall:
    for remotename in connection.nlst():         # попробовать удалить все файлы на сервере,
        try:                                     # чтобы убрать старые файлы
            print 'deleting remote', remotename
            connection.delete(remotename)
        except:
            print 'cannot delete remote', remotename

count = 0
localfiles = os.listdir(localdir)                # посылать все локальные файлы
                                                 # listdir() отрезает путь к каталогу
for localname in localfiles:
    localpath = os.path.join(localdir, localname)
    print 'uploading', localpath, 'to', localname
    if localname[-4:] == 'html' or localname[-3:] == 'txt':
        # передача в ascii-режиме
        localfile = open(localpath, 'r')
        connection.storlines('STOR ' + localname, localfile)
    else:
        # передача в двоичном режиме
        localfile = open(localpath, 'rb')
        connection.storbinary('STOR ' + localname, localfile, 1024)
    localfile.close()
    count = count+1

connection.quit()
print 'Done:', count, 'files uploaded.'
```

Как и сценарий зеркальной загрузки с сервера, эта программа иллюстрирует ряд новых интерфейсов и приемов написания сценариев FTP:

Удаление всех файлов на сервере

Как и в сценарии зеркального копирования, загрузка на сервер начинается с запроса необходимости удаления всех файлов в целевом каталоге сервера перед копированием туда новых файлов. Эта опция `cleanall` полезна, если какие-то файлы удалены из локальной копии каталога у клиента – удаленные файлы останутся в экземпляре на сервере, если сначала не удалить там все файлы. Чтобы реализовать зачистку сервера, этот сценарий просто получает список всех файлов в каталоге на сервере с помощью метода FTP `nlst` и поочередно удаляет их с помощью метода FTP `delete`. Если есть право на удаление, каталог будет очищен (права доступа к файлам зависят от учетной записи, по которой производится регистрация при подключении к серверу). Когда происходит удаление, мы уже находимся в целевом удаленном каталоге, поэтому не требуется указывать путь к каталогу перед именем файла.

Запись всех локальных файлов

Чтобы применить операцию загрузки на сервер к каждому файлу локального каталога, мы получаем список локальных имен файлов стандартным вызовом `os.listdir` и обеспечиваем добавление пути к локальному каталогу в начало каждого имени файла с помощью вызова `os.path.join`. Вспомните, что `os.listdir` возвращает имена файлов без путей каталогов, а исходный каталог может отличаться от каталога исполнения сценария, если он передан в командной строке.

Загрузка на сервер в текстовом режиме

Этот сценарий могут выполнять клиенты в Windows и Unix-подобных системах, поэтому текстовые файлы должны обрабатываться особым способом. Как и при зеркальной загрузке с сервера, этот сценарий устанавливает текстовый или двоичный режим передачи в соответствии с расширением имени файла – файлы HTML и текстовые перемещаются в текстовом режиме FTP. Мы уже знакомы с методом `storbinary` объекта FTP, применяемым для передачи файлов в двоичном режиме – на удаленном сайте оказывается точная, байт в байт, копия файла.

Передача в текстовом режиме действует почти так же: метод `storlines` принимает строку команды FTP и объект локального файла (или типа файла), открытый в текстовом режиме, и просто копирует каждую строку локального файла в одноименный файл на удаленной машине. Как обычно, если выполнять этот сценарий под Windows, открытие входного файла в текстовом режиме «r» означает, что последовательности символов конца строки в стиле DOS `\r\n` при чтении строк преобразуются в символ `\n`. Если сценарий выполняется в Unix или Linux, строки уже оканчиваются одиночным `\n`, поэтому такого преобразования не происходит. В итоге данные читаются переносимым образом, с представлением конца строки в виде символа `\n`. Двоичные файлы открываются в режиме «rb», чтобы подавить такое автоматическое преобразование (нам не нужно, чтобы байты, значение которых случайно совпадает с `\r`, таинственным образом исчезали при чтении под Windows).[1]

Как и в сценарии зеркальной загрузки с сервера, эта программа обходит все файлы, которые должны быть переданы (в данном случае в перечне локального каталога), и поочередно передает их – в текстовом или двоичном режиме в зависимости от имени файла. Вот команда, которой я пользуюсь для отправки целиком моего сайта с переносного компьютера под Windows 98 на удаленный Unix-сервер моего Интернет-провайдера за один шаг:

```
C:\Stuff\Website\public_html>python %X%\Internet\Ftp\uploadflat.py
Please enter password for starship.python.net:
Clean remote directory first?
connecting...
uploading .\LJsuppcover.jpg to LJsuppcover.jpg
uploading .\PythonPowered.gif to PythonPowered.gif
uploading .\PythonPoweredAnim.gif to PythonPoweredAnim.gif
uploading .\PythonPoweredSmall.gif to PythonPoweredSmall.gif
uploading .\Pywin.gif to Pywin.gif
uploading .\UPDATES to UPDATES
uploading .\about-hopl.html to about-hopl.html
uploading .\about-lp.html to about-lp.html
uploading .\about-pp-japan.html to about-pp-japan.html
...
...здесь удалены строки...
...
uploading .\trainingCD.GIF to trainingCD.GIF
uploading .\uk-1.jpg to uk-1.jpg
uploading .\uk-2.jpg to uk-2.jpg
uploading .\uk-3.jpg to uk-3.jpg
uploading .\uploadflat.py to uploadflat.py
```

[1] Технически метод Python `storlines` автоматически посылает на сервер все строки с последовательностью перевода строки `\r\n` независимо от того, что он получает от метода `readline` локального файла (`\n` или `\r\n`). По этой причине самым важным отличием при загрузке на сервер является использование «rb» для двоичного режима и метода `storlines` для текста. За деталями обратитесь к модулю *ftplib.py* в каталоге исходного кода библиотеки Python.

```
uploading .\whatsnew.html to whatsnew.html
uploading .\whatsold.html to whatsold.html
uploading .\xlate-lp.html to xlate-lp.html
Done: 131 files uploaded.
```

Как и в примере с зеркальным копированием, я обычно выполняю эту команду из локального каталога, где хранятся мои веб-файлы, и передаю в Python полный путь к сценарию. Если я выполняю этот сценарий на Linux-сервере *starship*, он действует так же, но пути к сценарию и каталогу моих веб-файлов различны. Если вы решите очистить удаленный каталог перед отправкой в него файлов, то также получите серию сообщений «deleting remote...» перед строками «uploading...»:

```
...
deleting remote uk-3.jpg
deleting remote whatsnew.html
deleting remote whatsold.html
deleting remote xlate-lp.html
deleting remote uploadflat.py
deleting remote ora-lp-france.gif
deleting remote LJsuppcover.jpg
deleting remote sonyz505js.gif
deleting remote pic14.html
...
```

Загрузка на сервер вместе с подкаталогами

Возможно, самым большим ограничением рассмотренных сценариев загрузки вебсайта с сервера и на сервер является допущение, что каталог сайта является плоским (отсюда их названия), то есть оба сценария передают только простые файлы и не обрабатывают вложенные каталоги внутри веб-каталога, который пересылается.

Для моих целей это разумное ограничение. Для простоты я избегаю вложенных подкаталогов и храню свой домашний сайт как простой каталог файлов. Для других сайтов (включая тот, который я держу на машине *starship*) сценариями пересылки файлов проще пользоваться, если они попутно также автоматически пересылают подкаталоги.

Оказывается, поддержка передачи подкаталогов осуществляется достаточно просто – требуется добавить только немного рекурсии и вызовы для создания удаленных каталогов. Сценарий загрузки на сервер примера 11.14 расширяет тот, который мы только что видели, осуществляя передачу всех подкаталогов, вложенных в пересылаемый каталог. Более того, он рекурсивно пересылает каталоги, находящиеся внутри подкаталогов – все дерево каталогов, содержащееся внутри передаваемого каталога верхнего уровня, загружается в целевой каталог удаленного сервера.

Пример 11.14. PP2E\Internet\Ftp\uploadall.py

```
#!/bin/env python
########################################################################
# С помощью ftp загрузить все файлы локального каталога на удаленный сайт/каталог;
# эта версия также поддерживает загрузку вложенных каталогов, но не опцию cleanall
# (для этого требуется анализ листингов ftp, чтобы определить удаленные каталоги и т. п.);
# для загрузки подкаталогов используется os.path.isdir(path) для проверки, является ли
# локальный файл в действительности каталогом, FTP().mkd(path) для создания каталога
# на удаленной машине (заключенный в try на случай, если каталог уже существует) и рекурсия
# для отправки всех файлов/каталогов внутри вложенного подкаталога. См. также: uploadall-2.py,
# который не предполагает существования верхнего удаленного каталога.
########################################################################
```

```
import os, sys, ftplib
from getpass import getpass

remotesite   = 'home.rmi.net'              # пересылка с pc или starship на rmi.net
topremotedir = 'public_html'
remoteuser   = 'lutz'
remotepass   = getpass('Please enter password for %s: ' % remotesite)
toplocaldir  = (len(sys.argv) > 1 and sys.argv[1]) or '.'

print 'connecting...'
connection = ftplib.FTP(remotesite)        # соединиться с сайтом ftp
connection.login(remoteuser, remotepass)   # регистрироваться как пользователь/пароль
connection.cwd(topremotedir)               # перейти в каталог, куда нужно копировать
                                           # предполагается существование
def uploadDir(localdir):
    global fcount, dcount
    localfiles = os.listdir(localdir)
    for localname in localfiles:
        localpath = os.path.join(localdir, localname)
        print 'uploading', localpath, 'to', localname
        if os.path.isdir(localpath):
            # рекурсия в подкаталоги
            try:
                connection.mkd(localname)
                print localname, 'directory created'
            except:
                print localname, 'directory not created'
            connection.cwd(localname)
            uploadDir(localpath)
            connection.cwd('..')
            dcount = dcount+1
        else:
            if localname[-4:] == 'html' or localname[-3:] == 'txt':
                # передача в режиме ascii
                localfile = open(localpath, 'r')
                connection.storlines('STOR ' + localname, localfile)
            else:
                # передача в двоичном режиме
                localfile = open(localpath, 'rb')
                connection.storbinary('STOR ' + localname, localfile, 1024)
            localfile.close()
            fcount = fcount+1

fcount = dcount = 0
uploadDir(toplocaldir)
connection.quit()
print 'Done:', fcount, 'files and', dcount, 'directories uploaded.'
```

Этот сценарий, как и сценарий для плоской загрузки, может выполняться на любой машине с Python и сокетами и осуществлять загрузку на любую машину, на которой работает сервер FTP; я выполняю его на своем переносном компьютере и на *starship* через telnet для загрузки сайтов к моему Интернет-провайдеру.

В целях экономии места я оставляю более глубокое изучение этого варианта в качестве упражнения для читателя. Однако дам два кратких указания:

- Суть дела здесь в проверке `os.path.isdir`, осуществляемой в начале. Если эта проверка обнаруживает каталог в текущем локальном каталоге, создаем на удаленной машине одноименный каталог с помощью `connection.mkd`, входим в него с по-

мощью `connection.cwd` и спускаемся в подкаталог на локальной машине. Как и все методы объекта FTP, `mkd` и `cwd` посылают команды FTP удаленному серверу. Выйдя из локального подкаталога, мы выполняем удаленный `cwd('..')`, чтобы подняться в удаленный родительский каталог, и продолжаем. Остальная часть сценария примерно совпадает с первоначальным вариантом.

- Обратите внимание, что этот сценарий обрабатывает только загрузку дерева каталогов *на сервер*; рекурсивная отправка обычно более полезна, чем рекурсивное получение, если поддерживать свои сайты на локальном PC и периодически отправлять их на сервер, как это делаю я. Если нужно также загрузить с сервера (сделать зеркальную копию) сайта с подкаталогами, посмотрите сценарии копирования в каталоге Tools дистрибутива исходного кода Python (в данное время это *Tools/scripts/ftpmirror.py*). Дополнительной работы требуется немного, но необходимо произвести разбор вывода команды удаленного листинга, чтобы найти на сервере каталоги, что опущено здесь ввиду достаточной сложности. По той же причине показанный здесь сценарий рекурсивной загрузки на сервер не поддерживает имеющуюся в оригинале опцию очистки удаленного каталога – эта функция также потребовала бы анализа перечней файлов на сервере.

Для полноты представления взгляните также на версию этого сценария *uploadall-2.py* в дистрибутиве примеров. Она аналогична данной, но не предполагает изначального существования удаленного каталога верхнего уровня.

Обработка электронной почты Интернета

Некоторые другие наиболее часто встречающиеся Интернет-протоколы верхнего уровня предназначены для чтения и отправки сообщений электронной почты: POP и IMAP для получения почты с серверов,[1] SMTP для отправки новых сообщений, а дополнительные спецификации, например `rfc822`, определяют содержимое и формат сообщений e-mail. При использовании стандартных почтовых средств обычно не требуется знать о существовании этих акронимов, но внутри таких программ, как Microsoft Outlook, при обработке ваших запросов происходит обмен данными с серверами POP и SMTP.

Как и FTP, электронная почта в конечном счете состоит из форматированных команд и потоков байтов, передаваемых через сокеты и порты (порт 110 для POP; 25 для SMTP). И так же, как для FTP, в Python есть стандартные модули, которые упрощают все стороны обработки электронной почты. В этом разделе мы исследуем интерфейсы POP и SMTP для получения и отправки почты на серверы и интерфейсы `rfc822` для выделения информации из строк заголовков e-mail; другие интерфейсы электронной почты в Python аналогичны и описаны в справочном руководстве по библиотеке Python.

POP: чтение e-mail

Раньше я был старомоден в своих привычках. Признаюсь, вплоть до недавнего времени я предпочитал читать свою электронную почту, подключаясь к провайдеру по telnet и используя простой интерфейс командной строки. Конечно, для почты с вложениями, картинками и т. п. это не идеально, но переносимость впечатляет: поскольку telnet работает почти на всякой машине, подключенной к сети, я мог быстро и просто

[1] IMAP, или Internet Message Access Protocol, был разработан в качестве альтернативы POP, но используется сегодня не столь широко, поэтому в данной книге он не представлен. О деталях, касающихся поддержки IMAP, читайте в руководстве по библиотеке Python.

проверить свою почту, находясь в любой точке земного шара. С учетом того, что я зарабатываю себе на жизнь, разъезжая по свету и преподавая Python, такая широкая доступность была большим преимуществом.

Если ранее в этой главе вы уже видели разделы со сценариями для зеркального копирования сайтов, то знаете и мою историю жалоб на провайдера, поэтому я не стану повторять ее снова. Достаточно сказать, что и на этом фронте произошли изменения: когда мой провайдер отменил доступ по telnet, я лишился также доступа к электронной почте.[1] К счастью, и здесь на помощь пришел Python: создав сценарии Python для доступа к электронной почте, я снова могу читать и отправлять e-mail с любой машины, где есть Python и соединение с Интернетом. Python может быть таким же переносимым решением, как telnet.

Кроме того, я могу использовать эти сценарии в качестве альтернативы тем средствам, которые предлагает мне провайдер, например Microsoft Outlook. Я не большой любитель предоставлять контроль коммерческим продуктам больших компаний, а кроме того, такие средства, как Outlook, обычно загружают почту на PC и удаляют ее с почтового сервера после обращения к нему. В результате ваш почтовый ящик занимает мало места (и ваш провайдер доволен), но по отношению к странствующим продавцам Python это не очень вежливо – получив доступ к письму, вы уже не сможете сделать это повторно, кроме как с той машины, на которую оно первоначально было загружено. Если нужно посмотреть старое письмо, а под рукой нет вашего PC, вам не повезло.[2]

Следующие два сценария представляют возможное решение таких проблем переносимости и необходимости использования одной машины (с другими решениями мы познакомимся далее в этой и последующих главах). Первый из них, *popmail.py*, служит простой утилитой чтения почты, которая загружает и выводит содержимое каждого сообщения в почтовом ящике. Этот сценарий явно примитивен, но позволяет читать свою электронную почту с любой машины, на которой установлены Python и сокеты; кроме того, он оставляет почту на сервере в целости. Второй сценарий, *smtpmail.py*, служит одновременно для создания и отправки новых сообщений электронной почты.

Модуль конфигурации электронной почты

Однако прежде чем перейти к этим сценариям, рассмотрим общий модуль, который они импортируют и используют. Модуль примера 11.15 используется для настройки параметров e-mail для конкретного пользователя. Это просто ряд присваиваний, используемых во всех почтовых программах, имеющихся в этой книге. Выделение этих установок конфигурации в отдельный модуль упрощает настройку представленных в книге почтовых программ для использования конкретным пользователем.

Если вы хотите пользоваться какими-либо почтовыми программами из этой книги для самостоятельной обработки почты, исправьте присвоения, производимые в этом модуле, так чтобы они отражали ваши серверы, названия учетных записей и т. д. (в представленном виде они соответствуют моим учетным записям электронной почты). Не все настройки из этого модуля используются в последующих двух сценариях;

---

[1] При потере telnet мой почтовый ящик и сайт оказались бездействующими на несколько недель, и я утратил навсегда архив из тысяч сообщений, накопленный в течение года. Такие отключения особенно болезненны, если ваш заработок существенно зависит от контактов через электронную почту и Сеть, но эту историю, девочки и мальчики, я расскажу вам в следующий раз.

[2] Автор, видимо, действительно старомоден в своих привычках. Настройка, позволяющая сохранять почту на сервере, присутствует практически в любом почтовом клиенте. В том числе и в Microsoft Outlook. Что, впрочем, нисколько не умаляет ценность данного примера. – *Примеч. ред.*

некоторые из них будут объяснены, когда мы вернемся к этому модулю в более поздних примерах.

Пример 11.15. PP2E\Internet\Email\mailconfig.py

```
##############################################################################################
# Сценарии e-mail берут имена серверов и другие параметры конфигурации e-mail
# из этого модуля: измените его так, чтобы он отражал имена ваших машин,
# подписи и т. п.; некоторые из них можно было бы передавать в командной строке;
##############################################################################################

#-----------------------------------------
# имя машины сервера SMTP (отправка)
#-----------------------------------------

smtpservername = 'smtp.rmi.net'     # или starship.python.net, 'localhost'

#-----------------------------------------
# POP3 email server machine, user (retrieve)
#-----------------------------------------

popservername  = 'pop.rmi.net'      # или starship.python.net, 'localhost'
popusername    = 'lutz'             # пароль запрашивается во время исполнения

#-----------------------------------------
# локальный файл, где pymail сохраняет pop-почту
# PyMailGui запрашивает во всплывающем диалоге
#-----------------------------------------

savemailfile   = r'c:\stuff\etc\savemail.txt'       # в PyMailGui используется диалог

#-----------------------------------------------------------------
# PyMailGui: необязательное имя локального текстового файла с одной
# строкой - вашим паролем pop; если он пуст или не читается, пароль
# запрашивается при выполнении; пароль не шифруется, поэтому оставляйте его пустым
# на машинах общего пользования; PyMailCgi и pymail всегда спрашивают пароль при запуске.
#-----------------------------------------------------------------

poppasswdfile  = r'c:\stuff\etc\pymailgui.txt'      # установить в '', чтобы запрашивался

#-----------------------------------------------------------------
# Личные данные, используемые PyMailGui для заполнения форм;
# sig - может быть блоком в тройных кавычках, игнорируется, если строка пустая;
# addr - если не пуста, используется в качестве начального значения
# поля "From" либо с переменным успехом пытается угадать From для ответа;
#-----------------------------------------------------------------

myaddress   = 'lutz@rmi.net'
mysignature = '--Mark Lutz  (http://rmi.net/~lutz)  [PyMailGui 1.0]'
```

Модуль чтения почты с сервера POP

Переходим к чтению e-mail в Python: сценарий примера 11.16 пользуется стандартным модулем Python `poplib`, реализацией интерфейса клиента POP – Post Office Protocol (почтовый протокол). POP представляет собой корректно определенный способ получения e-mail с сервера через сокеты. Данный сценарий соединяется с сервером POP, реализуя простой, но переносимый инструмент загрузки и отображения электронной почты.

Пример 11.16. PP2E\Internet\Email\popmail.py

```
#!/usr/local/bin/python
```

```
#########################################################################################################
# Использует модуль POP3 почтового интерфейса Python для просмотра сообщений почтовой учетной
# записи pop; это простой листинг - см. в pymail.py клиент с большим числом функций
# взаимодействия с пользователем и в smtpmail.py сценарий отправки почты; pop используется
# для извлечения почты и выполняется на сокете с портом номер 110 на машине сервера,
# но модуль Python poplib скрывает все детали протокола; для отправки почты используйте модуль
# smtplib (или os.popen('mail...'). см. также: читалка unix mailfile в рамках структуры App.
#########################################################################################################

import poplib, getpass, sys, mailconfig

mailserver = mailconfig.popservername            # например: 'pop.rmi.net'
mailuser   = mailconfig.popusername              # например: 'lutz'
mailpasswd = getpass.getpass('Password for %s?' % mailserver)

print 'Connecting...'
server = poplib.POP3(mailserver)
server.user(mailuser)                     # соединиться, зарегистрироваться на почтовом сервере
server.pass_(mailpasswd)                   # pass является зарезервированным словом

try:
    print server.getwelcome()              # вывести возвращенное приветственное сообщение
    msgCount, msgBytes = server.stat()
    print 'There are', msgCount, 'mail messages in', msgBytes, 'bytes'
    print server.list()
    print '-'*80
    if sys.platform[:3] == 'win': raw_input()    # в Windows getpass особенный
    raw_input('[Press Enter key]')

    for i in range(msgCount):
        hdr, message, octets = server.retr(i+1)  # octets считает байты
        for line in message: print line          # извлечь и вывести всю почту
        print '-'*80                             # почтовый ящик заблокирован до quit
        if i < msgCount - 1:
           raw_input('[Press Enter key]')
finally:                                         # разблокировать почтовый ящик
    server.quit()                                # иначе - заперт до тайм-аута
print 'Bye.'
```

Несмотря на свою примитивность, этот сценарий иллюстрирует основы чтения электронной почты в Python. Чтобы установить соединение с почтовым сервером, мы сначала создаем экземпляр объекта `poplib.POP3`, передавая ему имя сервера:

```
server = poplib.POP3(mailserver)
```

Если этот вызов не возбудил исключительную ситуацию, значит, мы соединились (через сокет) с POP-сервером, который слушает запросы на порту POP с номером 110 на машине, где находится наша учетная запись почты. Следующее, что необходимо сделать перед получением сообщений, это сообщить серверу имя пользователя и пароль; обратите внимание, что метод передачи пароля называется `pass_` – без символа подчеркивания в конце `pass` оказался бы зарезервированным словом и вызвал синтаксическую ошибку:

```
server.user(mailuser)      # соединиться, зарегистрирваться на почтовом сервере
server.pass_(mailpasswd)   # pass является зарезервированным словом
```

Для простоты и относительной безопасности этот сценарий всегда запрашивает пароль в интерактивном режиме; чтобы ввести, но не отображать на экране строку пароля, введенную пользователем, используется модуль `getpass`, знакомый по разделу этой главы, посвященному FTP.

Сообщив серверу имя пользователя и пароль, мы можем с помощью метода `stat` получить информацию о почтовом ящике (количество сообщений, суммарное количество байтов в сообщениях) и получить конкретное письмо с помощью метода `retr` (передав номер сообщения начиная с 1):

```
msgCount, msgBytes = server.stat()
hdr, message, octets = server.retr(i+1)    # octets считает байты
```

Закончив работу с почтой, закрываем соединение с почтовым сервером, вызывая метод `quit` объекта POP:

```
server.quit()                              # иначе заперт до тайм-аута
```

Обратите внимание, что этот вызов помещен в раздел `finally` предложения `try`, охватывающего основную часть сценария. Для борьбы со сложностями, связанными с изменениями данных, POP-серверы блокируют почтовый ящик на время между первым соединением и закрытием соединения (или истечением устанавливаемого системой тайм-аута произвольной длительности). Так как метод POP `quit` тоже разблокирует почтовый ящик, важно выполнить его перед завершением работы независимо от того, была ли во время обработки электронной почты возбуждена исключительная ситуация. Заключив действия в конструкцию `try/finally`, мы гарантируем вызов сценарием `quit` при выходе, чтобы разблокировать почтовый ящик и сделать его доступным другим процессам (например, для доставки входящей почты).

Вот пример работы сценария popmail, во время которой выводятся два сообщения из моего почтового ящика на машине *pop.rmi.net* (доменное имя почтового сервера на *rmi.net*, заданное в модуле `mailconfig`):

```
C:\...\PP2E\Internet\Email>python popmail.py
Password for pop.rmi.net?
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <4860000073ed6c39@chevalier>
There are 2 mail messages in 1386 bytes
('+OK 2 messages (1386 octets)', ['1 744', '2 642'], 14)
-------------------------------------------------------------------------------

[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:13:33 2000)
X-From_: lumber.jack@TheLarch.com  Wed Jul 12 16:10:28 2000
Return-Path: <lumber.jack@TheLarch.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA21434
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:10:27 -0600 (MDT)
From: lumber.jack@TheLarch.com
Message-Id: <200007122210.QAA21434@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:03:59 2000
Subject: I'm a Lumberjack, and I'm okay
X-Mailer: PyMailGui Version 1.0 (Python)

I cut down trees, I skip and jump,
I like to press wild flowers...

-------------------------------------------------------------------------------

[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:13:54 2000)
X-From_: lutz@rmi.net  Wed Jul 12 16:12:42 2000
```

```
Return-Path: <lutz@chevalier.rmi.net>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA24093
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:12:37 -0600 (MDT)
Message-Id: <200007122212.QAA24093@chevalier.rmi.net>
From: lutz@rmi.net
To: lutz@rmi.net
Date: Wed Jul 12 16:06:12 2000
Subject: testing
X-Mailer: PyMailGui Version 1.0 (Python)

Testing Python mail tools.

------------------------------------------------------------------------------

Bye.
```

Этот интерфейс предельно прост: после соединения с сервером он выводит полный необработанный текст очередного сообщения, делая после каждого остановку и ожидая нажатия клавиши <Enter>. Для ожидания нажатия клавиши между выводом сообщений вызывается встроенная функция `raw_input`.[1] Благодаря этой паузе не происходит слишком быстрой прокрутки сообщений на экране. Чтобы зрительно выделить сообщения, они также разделяются строками черточек. Можно было бы сделать отображение более изощренным (например, в более поздних примерах мы будем выделять части сообщений с помощью модуля `rfc822`), но здесь мы просто выводим целиком отправленное сообщение.

Если внимательно посмотреть на текст этих почтовых сообщений, то можно заметить, что они были отправлены другой программой, имя которой PyMailGui (мы познакомимся с ней ближе к концу главы). Строка заголовка «X-Mailer», если она присутствует, обычно идентифицирует программу-отправитель. На самом деле есть множество других строк заголовка, которые могут пересылаться в тексте сообщения. Например, заголовки «Received:» трассируют те машины, через которые прошло сообщение по пути к целевому почтовому ящику. Поскольку `popmail` выводит весь текст сообщения в необработанном виде, здесь видны все заголовки, но в почтовых GUI, ориентированных на конечного пользователя, таких как Outlook, по умолчанию можно видеть лишь некоторые из них.

Прежде чем двинуться дальше, я должен отметить, что этот сценарий не удаляет почту с сервера. Почта лишь загружается и выводится на экран и будет снова показана при следующем запуске сценария (если не удалить ее другими средствами). Для того чтобы действительно удалить почту, требуется вызывать другие методы (например, `server.dele(msgnum)`), но такие возможности лучше отложить до момента разработки более интерактивных почтовых инструментов.

SMTP: отправка электронной почты

Среди хакеров бытует высказывание, что каждая полезная компьютерная программа по достижении известного уровня сложности включает в себя возможность отправки электронной почты. Независимо от того, оправдывается ли эта старинная мудрость на практике, возможность автоматического создания в программе электронных писем является мощным инструментом.

---

[1] Дополнительный `raw_input` вставляется только в Windows, чтобы очистить поток от повреждений, наносимых вызовом `getpass`; см. примечание на эту тему в разделе этой главы, посвященном FTP.

Например, системы тестирования могут автоматически посылать по электронной почте сообщения об отказах, программы с пользовательским интерфейсом могут посылать заказы на поставки, и т. д. Кроме того, переносимый почтовый сценарий Python можно использовать для отправки сообщений с любого компьютера, на котором есть Python и соединение с Интернетом. Независимость от почтовых программ типа Outlook весьма привлекательна, когда вы зарабатываете себе на жизнь, разъезжая по свету и обучая Python на самых разнообразных компьютерах.

К счастью, отправить почту из сценария Python столь же просто, как и прочесть ее. В действительности существует не менее четырех способов сделать это:

Вызов os.popen для запуска почтовой программы командной строки

На некоторых системах отправить e-mail из сценария можно с помощью вызова следующего вида:

```
os.popen('mail -s "xxx" a@b.c', 'w').write(text)
```

Как мы уже видели ранее, `popen` выполняет командную строку, переданную ему в качестве первого аргумента, и возвращает объект типа файла, соединенный с командой. При открытии в режиме «w» происходит соединение со стандартным входным потоком команды – в данном случае текст нового почтового сообщения записывается в `mail` – стандартную программу командной строки Unix. Результат получается таким же, каким он был бы при интерактивном запуске `mail`, но достигается внутри работающего сценария Python.

Запуск программы sendmail

Программа с открытым исходным кодом `sendmail` предоставляет еще один способ запуска электронной почты из сценария. Если она установлена на вашей системе и настроена, можно запускать ее с помощью таких средств Python, как вызов `os.popen` из предыдущего параграфа.

Применение стандартного модуля Python smtplib

В стандартную библиотеку Python включена поддержка на стороне клиента интерфейса к SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты) – стандарту Интернета более высокого уровня для отправки почты через сокеты. Подобно модулю `poplib` из предыдущего раздела, `smtplib` скрывает все детали, касающиеся сокетов и протокола, и может применяться для отправки почты с любой машины, где есть Python и соединение с Интернетом на базе сокетов.

Получение и использование пакетов и средств сторонних разработчиков

В библиотеке ПО с открытым кодом есть другие средства, предоставляющие пакеты Python для обработки почты на более высоком уровне (имеются на *http://www.python.org*). Большинство из них основано на одной из трех вышеперечисленных технологий.

Из всех четырех вариантов наиболее переносимым и мощным является `smtplib`. Запуск почтовой программы с помощью `popen` обычно действует только на Unix-подобных платформах, но не в Windows (требуется наличие почтовой программы командной строки). И хотя программа `sendmail` обладает значительной мощью, она также имеет некоторый уклон в сторону Unix, сложна и даже не на всех Unix-подобных машинах может быть установлена.

Напротив, модуль `smtplib` работает на любой машине, где есть Python и соединение с Интернетом, в том числе под Unix, Linux и Windows. Кроме того, SMTP позволяет в значительной мере управлять форматированием и маршрутизацией e-mail. Поскольку можно утверждать, что это лучший вариант отправки почты из сценариев Python, рассмотрим простую почтовую программу, иллюстрирующую его интерфейсы.

Сценарий Python, показанный в примере 11.17, предназначен для использования из интерактивной командной строки; он читает новое почтовое сообщение, вводимое пользователем, и отправляет почту по SMTP с помощью модуля Python `smtplib`.

Пример 11.17. PP2E\Internet\Email\smtpmail.py

```
#!/usr/local/bin/python
#########################################################################
# Отправка сообщений e-mail с помощью модуля Python почтового интерфейса SMTP;
# это простой сценарий одноразовой отправки - см. pymail, PyMailGui и PyMailCgi
# в качестве клиентов с большими возможностями взаимодействия с пользователем,
# и popmail.py как сценарий получения почты;
#########################################################################

import smtplib, string, sys, time, mailconfig
mailserver = mailconfig.smtpservername      # например: starship.python.net

From = string.strip(raw_input('From? '))    # например: lutz@rmi.net
To   = string.strip(raw_input('To?   '))    # например: python-list@python.org
To   = string.split(To, ';')                # допускается список получателей
Subj = string.strip(raw_input('Subj? '))

# предпослать стандартные заголовки
date = time.ctime(time.time())
text = ('From: %s\nTo: %s\nDate: %s\nSubject: %s\n\n' % (From, string.join(To, ';'), date, Subj))

print 'Type message text, end with line=(ctrl + D or Z)'
while 1:
    line = sys.stdin.readline()
    if not line:
        break                                # выход по ctrl-d/z
  # if line[:4] == 'From':
  #     line = '>' + line                    # сервер осуществит преобразование
    text = text + line

if sys.platform[:3] == 'win': print
print 'Connecting...'
server = smtplib.SMTP(mailserver)            # соединение без регистрации
failed = server.sendmail(From, To, text)
server.quit()
if failed:                                  # smtplib может также возбуждать исключительные
    print 'Failed recipients:', failed       # ситуации, но они здесь не обрабатываются
else:
    print 'No errors.'
print 'Bye.'
```

Большая часть этого сценария составляет *интерфейс пользователя* – вводится адрес отправителя («From»), один или более адресов получателя («To», разделенные символом «;») и строка темы сообщения. Дата отправки берется из стандартного модуля Python `time`, стандартные строки заголовка форматируются, и цикл `while` читает строки сообщения, пока пользователь не введет символ конца файла (<Ctrl>+<Z> в Windows, <Ctrl>+<D> в Linux).

В остальной части сценария происходят все чудеса SMTP: для отправки почты по SMTP нужно выполнить следующие два типа вызовов:

```
server = smtplib.SMTP(mailserver)
```

Создать экземпляр объекта SMTP, передав ему имя сервера SMTP, который изначально отправит сообщение. Если при этом не возникнет исключительной ситуации, значит, при возврате из вызова вы соединены с SMTP-сервером через сокет.

```
failed = server.sendmail(From, To, text)
```

Вызвать метод `sendmail` объекта SMTP с передачей ему адреса отправителя, одного или более адресов получателя и собственно текста сообщения со всеми строками стандартных почтовых заголовков, какие вы зададите. Для надежности добавьте пустую строку между строками заголовков и телом сообщения.

Закончив передачу, вызовите метод `quit`, чтобы отключиться от сервера. Заметьте, что при неудаче метод `sendmail` может возбудить исключительную ситуацию или возвратить список адресов получателей, которые не были приняты; сценарий обрабатывает последний случай, но позволяет исключительным ситуациям убить сценарий с выводом сообщения Python об ошибке.

Отправка сообщений

Ладно, пошлем миру несколько сообщений. Сценарий `smtpmail` является одноразовым: при каждом запуске вы можете послать только одно новое сообщение. Как и большинство клиентских утилит этой главы, его можно запустить на любом компьютере с Python и соединением с Интернетом. Вот пример его выполнения под Windows 98:

```
C:\...\PP2E\Internet\Email>python smtpmail.py
From? Eric.the.Half.a.Bee@semibee.com
To?   lutz@rmi.net
Subj? A B C D E F G
Type message text, end with line=(ctrl + D or Z)
Fiddle de dum, Fiddle de dee,
Eric the half a bee.

Connecting...
No errors.
Bye.
```

Это письмо посылается по моему адресу (*lutz@rmi.net*), поэтому в конечном счете оно окажется в моем почтовом ящике у Интернет-провайдера, но лишь пройдя через неопределенное число машин Сети и сетевые соединения неизвестной длины. На нижнем уровне это сложно, но обычно Интернет «просто работает».

Обратите, однако, внимание на адрес «From» – он абсолютно фиктивный (по крайне мере, мне так кажется). Оказывается, обычно можно задать произвольный адрес «From», так как SMTP не проверяет его действительность (проверяется только общий формат). Далее, в отличие от POP, в SMTP нет понятия имени пользователя и пароля, поэтому отправителя установить труднее. Необходимо лишь передать e-mail на любую машину, где есть сервер, слушающий на порту SMTP, и не требуется учетной записи на этой машине. В данном случае *Eric.the.Half.a.Bee@semibee.com* вполне годится в качестве отправителя; с таким же успехом можно было бы задать *Marketing.Geek.From.Hell@spam.com*.

Теперь я скажу вам нечто, но только с целью образования: такой режим лежит в основе появления в ящике всего этого раздражающего почтового мусора без настоящего адреса отправителя.[1] Торговцы, помешанные на мысли разбогатеть с помощью Ин-

[1] Такую мусорную почту обычно называют спамом (*spam*), намекая на сценку из Monty Python, в которой люди пытаются заказать в ресторане завтрак, и их все время заглушает группа викингов, которые все громче и громче поют хором: «spam, spam, spam...» (нет, в самом деле). Хотя спам можно применять разными способами, данное применение отличается как от появления в примерах этой книги, так и от сильно превозносимого продукта питания (консервированный колбасный фарш).

тернета, рассылают рекламу по всем известным им адресам и не указывают действительный адрес «From», чтобы скрыть свои следы.

Обычно, конечно, вы должны использовать один и тот же адрес «To» в сообщении и вызове SMTP и указывать свой действительный почтовый адрес в качестве значения «From» (иначе люди не смогут ответить на ваше послание). Более того, за исключением случаев, когда вы дразните вашу «вторую половину», отправка фальшивого адреса явно нарушает правила добропорядочного поведения в Интернете. Запустим сценарий снова, чтобы отправить еще одно письмо с более политически корректными координатами:

```
C:\...\PP2E\Internet\Email>python smtpmail.py
From? lutz@rmi.net
To?   lutz@rmi.net
Subj? testing smtpmail
Type message text, end with line=(ctrl + D or Z)
Lovely Spam! Wonderful Spam!
Connecting...
No errors.
Bye.
```

После этого можно запустить тот инструмент электронной почты, который обычно используется для доступа к почтовому ящику, и проверить результат этих двух операций отправки; в почтовом ящике должны появиться два новых письма независимо от того, какой почтовый клиент используется для их просмотра. Однако, поскольку мы уже написали сценарий Python для чтения почты, используем его в качестве средства проверки – при запуске сценария `popmail` из последнего раздела в конце списка писем обнаруживаются два наших новых сообщения:

```
C:\...\PP2E\Internet\Email>python popmail.py
Password for pop.rmi.net?
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <c4050000b6ee6c39@chevalier>
There are 6 mail messages in 10941 bytes
('+OK 6 messages (10941 octets)', ['1 744', '2 642', '3 4456', '4 697', '5 3791'
, '6 611'], 44)
--------------------------------------------------------------------------------
...
...здесь опущен ряд строк...
...
[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:19:20 2000)
X-From_: Eric.the.Half.a.Bee@semibee.com  Wed Jul 12 16:16:31 2000
Return-Path: <Eric.the.Half.a.Bee@semibee.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA28647
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:16:30 -0600 (MDT)
From: Eric.the.Half.a.Bee@semibee.com
Message-Id: <200007122216.QAA28647@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:09:21 2000
Subject: A B C D E F G

Fiddle de dum, Fiddle de dee,
Eric the half a bee.

--------------------------------------------------------------------------------
```

```
[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:19:51 2000)
X-From_: lutz@rmi.net  Wed Jul 12 16:17:58 2000
Return-Path: <lutz@chevalier.rmi.net>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA00415
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:17:57 -0600 (MDT)
Message-Id: <200007122217.QAA00415@chevalier.rmi.net>
From: lutz@rmi.net
To: lutz@rmi.net
Date: Wed Jul 12 16:10:55 2000
Subject: testing smtpmail

Lovely Spam! Wonderful Spam!

------------------------------------------------------------------------------

Bye.
```

Другие способы злоупотреблений в Сети

Первым письмом здесь было то, которое мы послали с фиктивным адресом; вторым было более законное сообщение. Как и в адресах «From», в строках заголовков SMTP также допускает некоторый произвол. `smtpmail` автоматически добавляет строки заголовков «From:» и «To:» в текст сообщения с теми адресами, которые были переданы интерфейсу SMTP, но только в порядке услуги. Иногда, однако, нельзя узнать и *кому* было послано письмо – чтобы скрыть круг получателей, спаммеры могут воспользоваться фокусами со слепыми копиями «Bcc» или содержимым заголовков в тексте сообщения.

Например, если изменить `smtpmail` так, чтобы не создавалась автоматически строка заголовка «To:» с теми же адресами, которые передаются вызову интерфейса SMTP, можно вручную ввести заголовок «To:», отличающийся от реального адреса отправки:

```
C:\...\PP2E\Internet\Email>python smtpmail-noTo.py
From? Eric.the.Half.a.Bee@semibee.com
To?   lutz@starship.python.net
Subj? a b c d e f g
Type message text, end with line=(ctrl + D or Z)
To: nobody.in.particular@marketing.com
Fiddle de dum, Fiddle de dee,
Eric the half a bee.
Connecting...
No errors.
Bye.
```

В некоторых отношениях адреса «From» и «To» в вызове метода отправки и заголовке сообщения аналогичны адресу на конверте и письму в конверте. Первое используется для пересылки, а второе представляет то, что видит читатель. Здесь я указал в качестве адреса «To» мой почтовый ящик на сервере *starship.python.net*, но ввел вручную фиктивное имя в строку заголовка «To:»; первый адрес определяет действительное место доставки сообщения. Если запустить на *starship* через telnet почтовую утилиту командной строки, то будут обнаружены два фальшивых письма – одно с неверным «From:», а другое еще и с неверным «To:», которое мы только что послали:

```
[lutz@starship lutz]$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/home/crew/lutz/Mailbox": 22 messages 12 new 22 unread
```

```
...еще строки...
>N 21 Eric.the.Half.a.Bee@  Thu Jul 13 20:22  20/789   "A B C D E F G"
 N 22 Eric.the.Half.a.Bee@  Thu Jul 13 20:26  19/766   "a b c d e f g"

& 21
Message 21:
From Eric.the.Half.a.Bee@semibee.com Thu Jul 13 20:21:18 2000
Delivered-To: lutz@starship.python.net
From: Eric.the.Half.a.Bee@semibee.com
To: lutz@starship.python.net
Date: Thu Jul 13 14:15:55 2000
Subject: A B C D E F G

Fiddle de dum, Fiddle de dee,
Eric the half a bee.

& 22
Message 22:
From Eric.the.Half.a.Bee@semibee.com Thu Jul 13 20:26:34 2000
Delivered-To: lutz@starship.python.net
From: Eric.the.Half.a.Bee@semibee.com
Date: Thu Jul 13 14:20:22 2000
Subject: a b c d e f g
To: nobody.in.particular@marketing.com

Fiddle de dum, Fiddle de dee,
Eric the half a bee.
```

Если почтовая программа выделяет строку «To:», то такие письма при просмотре выглядят странно. Например, пошлем еще одно письмо на мой почтовый ящик на *rmi.net*:

```
C:\...\PP2E\Internet\Email>python smtpmail-noTo.py
From? Arthur@knights.com
To?   lutz@rmi.net
Subj? Killer bunnies
Type message text, end with line=(ctrl + D or Z)
To: you@home.com
Run away!  Run away! ...
Connecting...
No errors.
Bye.
```

Когда оно окажется в моем почтовом ящике на *rmi.net*, трудно будет что-либо сказать о его источнике или адресате при просмотре в Outlook или написанной на Python почтовой программе, которую мы увидим в конце этой главы (см. рис. 11.8 и 11.9). А его необработанный текст покажет только машины, через которые оно прошло.

Еще раз напомню – не делайте таких вещей без веских причин. Я показываю это только с целью проиллюстрировать строки заголовков (например, ниже мы добавим строку заголовка «X-mailer:», идентифицирующую программу отправки). Кроме того, чтобы остановить преступника, иногда нужно рассуждать так же, как он, – вам не многого удастся добиться в борьбе со спамом, если вы не поймете, как он создается. Чтобы написать автоматический фильтр спама, удаляющий мусор из входящей почты, нужно знать сигнальные признаки, которые следует искать в тексте сообщения. А фокусы с адресами «To:» могут оказаться полезными в контексте законных списков рассылки.

Но в действительности отправка электронной почты с фальшивыми строками «From:» и «To:» эквивалентна анонимным телефонным звонкам. Большинство почтовых программ даже не позволяет изменить строку «From:» и не делает отличия между адре-

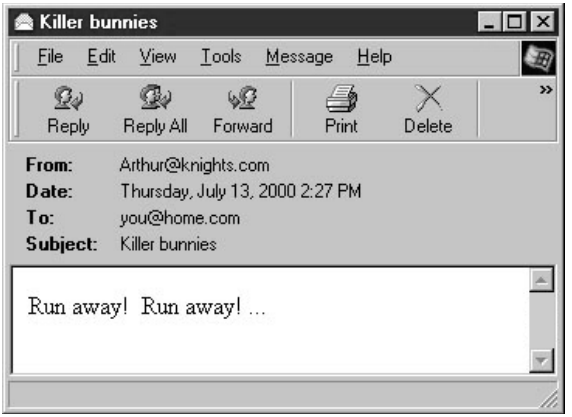


Рис. 11.8. Фальшивое письмо в Outlook

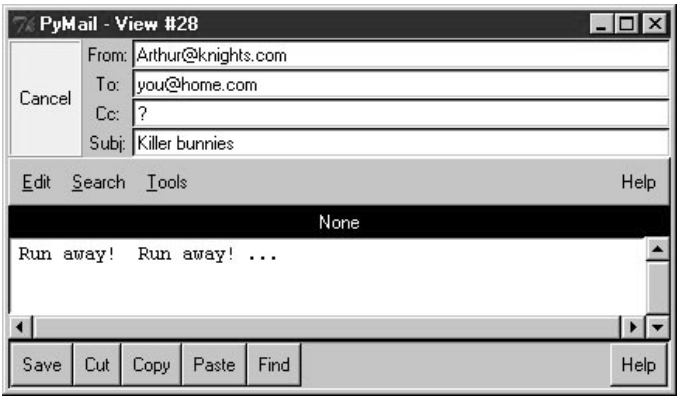


Рис. 11.9. Фальшивое письмо в почтовой программе Python (PyMailGui)

сом и строкой заголовка «To:», хотя SMTP широко открыт в этом отношении. Ведите себя хорошо, договорились?

Возвращаясь к общей картине Интернета

Итак, где же мы теперь находимся в абстрактной модели Интернета? Поскольку почта пересылается через сокеты (еще помните о них?), они лежат в основе всех этих получений и отправок электронной почты. Всякое прочтенное или написанное письмо в конечном итоге состоит из форматированных байтов, передаваемых между компьютерами в Сети через сокеты. Однако, как мы видели, в Python интерфейсы POP и SMTP скрывают все детали. Более того, сценарии, которые мы начали писать, скрывают даже интерфейсы Python и предоставляют интерактивные средства более высокого уровня.

Оба сценария, `popmail` и `smtpmail`, предоставляют переносимые инструменты электронной почты, но удобство их использования не совсем соответствует тому уровню, который принят в нынешние времена. В следующем разделе с помощью увиденного мы реализуем более интерактивное почтовое средство. В конце этого раздела, посвященного e-mail, мы также создадим почтовый GUI с использованием Tk, а в другой главе

построим интерфейс, основанный на веб. Однако все эти инструменты разнятся, главным образом, только в интерфейсе пользователя; каждый из них в конечном счете применяет рассмотренные нами почтовые модули для передачи текста почтового сообщения через Интернет посредством сокетов.

Почтовый клиент, выполняемый в командной строке

Теперь объединим вместе все, что мы узнали о получении и отправке e-mail, в простом, но функциональном средстве командной строки для электронной почты. Сценарий в примере 11.18 реализует интерактивный сеанс электронной почты – пользователи могут вводить команды для чтения, отправки и удаления электронных писем.

Пример 11.18. PP2E\Internet\Email\pymail.py

```
#!/usr/local/bin/python
###############################################################################################
# Простой клиент командной строки интерфейса e-mail на Python; использует модуль
# Python POP3 почтового интерфейса для просмотра сообщений учетной записи pop;
# использует модули rfc822 и StringIO для извлечения заголовков почтового сообщения;
###############################################################################################

import poplib, rfc822, string, StringIO

def connect(servername, user, passwd):
    print 'Connecting...'
    server = poplib.POP3(servername)
    server.user(user)            # соединиться и зарегистрироватся на почтовом сервере
    server.pass_(passwd)         # pass является зарезервированным словом
    print server.getwelcome()    # вывести возвращенное приветствие
    return server

def loadmessages(servername, user, passwd, loadfrom=1):
    server = connect(servername, user, passwd)
    try:
        print server.list()
        (msgCount, msgBytes) = server.stat()
        print 'There are', msgCount, 'mail messages in', msgBytes, 'bytes'
        print 'Retrieving:',
        msgList = []
        for i in range(loadfrom, msgCount+1):           # пусто, если низ >= верх
            print i,                                    # теперь получить почту
            (hdr, message, octets) = server.retr(i)     # сохранить текст в списке
            msgList.append(string.join(message, '\n'))  # оставить почту на сервере
        print
    finally:
        server.quit()                                   # разблокировать почтовый ящик
    assert len(msgList) == (msgCount - loadfrom) + 1    # номера сообщений  начинаются с 1
    return msgList

def deletemessages(servername, user, passwd, toDelete, verify=1):
    print 'To be deleted:', toDelete
    if verify and raw_input('Delete?')[:1] not in ['y', 'Y']:
        print 'Delete cancelled.'
    else:
        server = connect(servername, user, passwd)
        try:
            print 'Deleting messages from server.'
            for msgnum in toDelete:         # снова соединиться для удаления почты
                server.dele(msgnum)         # ящик заблокирован до quit()
```

```
        finally:
            server.quit()

def showindex(msgList):
    count = 0
    for msg in msgList:                     # отрезать и показать некоторые заголовки
        strfile = StringIO.StringIO(msg)    # пусть строка выглядит, как файл
        msghdrs = rfc822.Message(strfile)   # разобрать почтовые заголовки в словарь
        count   = count + 1
        print '%d:\t%d bytes' % (count, len(msg))
        for hdr in ('From', 'Date', 'Subject'):
            try:
                print '\t%s=>%s' % (hdr, msghdrs[hdr])
            except KeyError:
                print '\t%s=>(unknown)' % hdr
            #print '\n\t%s=>%s' % (hdr, msghdrs.get(hdr, '(unknown)'))
        if count % 5 == 0:
            raw_input('[Press Enter key]')  # pause after each 5

def showmessage(i, msgList):
    if 1 <= i <= len(msgList):
        print '-'*80
        print msgList[i-1]           # вывести целиком - заголовки+текст
        print '-'*80                 # чтобы получить только текст, вызовите file.read()
    else:                            # после чтения rfc822.Message строк заголовка
        print 'Bad message number'

def savemessage(i, mailfile, msgList):
    if 1 <= i <= len(msgList):
        open(mailfile, 'a').write('\n' + msgList[i-1] + '-'*80 + '\n')
    else:
        print 'Bad message number'

def msgnum(command):
    try:
        return string.atoi(string.split(command)[1])
    except:
        return -1   # предполагая, что это плохо

helptext = """
Available commands:
i     - index display
l n?  - list all messages (or just message n)
d n?  - mark all messages for deletion (or just message n)
s n?  - save all messages to a file (or just message n)
m     - compose and send a new mail message
q     - quit pymail
?     - display this help text
"""

def interact(msgList, mailfile):
    showindex(msgList)
    toDelete = []
    while 1:
        try:
            command = raw_input('[Pymail] Action? (i, l, d, s, m, q, ?) ')
        except EOFError:
            command = 'q'

        # завершение
```

```
        if not command or command == 'q':
            break

        # индекс
        elif command[0] == 'i':
            showindex(msgList)

        # список
        elif command[0] == 'l':
            if len(command) == 1:
                for i in range(1, len(msgList)+1):
                    showmessage(i, msgList)
            else:
                showmessage(msgnum(command), msgList)

        # сохранение
        elif command[0] == 's':
            if len(command) == 1:
                for i in range(1, len(msgList)+1):
                    savemessage(i, mailfile, msgList)
            else:
                savemessage(msgnum(command), mailfile, msgList)

        # удаление
        elif command[0] == 'd':
            if len(command) == 1:
                toDelete = range(1, len(msgList)+1)     # удалить все позднее
            else:
                delnum = msgnum(command)
                if (1 <= delnum <= len(msgList)) and (delnum not in toDelete):
                    toDelete.append(delnum)
                else:
                    print 'Bad message number'

        # почта
        elif command[0] == 'm':               # отправить новую почту через smtp
            try:                              # повторное использование существующего сценария
                execfile('smtpmail.py', {})   # запуск файла в собственном пространстве имен
            except:
                print 'Error - mail not sent' # не умирать, если умрет сценарий

        elif command[0] == '?':
            print helptext
        else:
            print 'What? -- type "?" for commands help'
    return toDelete

if __name__ == '__main__':
    import sys, getpass, mailconfig
    mailserver = mailconfig.popservername      # например: 'starship.python.net'
    mailuser   = mailconfig.popusername        # например: 'lutz'
    mailfile   = mailconfig.savemailfile       # например:  r'c:\stuff\savemail'
    mailpswd   = getpass.getpass('Password for %s?' % mailserver)

    if sys.platform[:3] == 'win': raw_input()  # очистить поток
    print '[Pymail email client]'
    msgList    = loadmessages(mailserver, mailuser, mailpswd) # загрузить все
    toDelete   = interact(msgList, mailfile)
    if toDelete: deletemessages(mailserver, mailuser, mailpswd, toDelete)
    print 'Bye.'
```

Нового здесь немного – просто сочетание логики интерфейса пользователя, уже знакомых нам инструментов и некоторых новых приемов:

Загрузка

Этот клиент загружает с сервера всю электронную почту в находящийся в оперативной памяти список Python только один раз, при начальном запуске; чтобы получить вновь поступившую почту, необходимо завершить программу и запустить ее заново.

Сохранение

По требованию `pymail` сохраняет исходный текст выбранного сообщения в локальном файле, имя которого указано в модуле `mailconfig`.

Удаление

Теперь, наконец, поддерживается удаление почты с сервера по соответствующему запросу: в `pymail` письма для удаления выбираются по номерам, но все же физически удаляются с сервера только при выходе и только при подтверждении операции. Благодаря удалению только при выходе из программы удается избежать изменения номеров почтовых сообщений во время сеанса – в POP удаление почты из середины списка ведет к уменьшению номеров всех сообщений, следующих за тем, которое удаляется. Так как `pymail` кэширует все сообщения в памяти, последующие операции с пронумерованными сообщениями в памяти могут быть неправильно применены, если осуществлять удаления незамедлительно.[1]

Разбор сообщений

Pymail по-прежнему выводит целиком исходный текст сообщения по командам вывода сообщений, но при выводе оглавления почтового ящика (mail index listing) отображаются только выбранные заголовки, выделенные из каждого сообщения. Для извлечения из сообщения заголовков используется модуль Python `rfc822`: вызов `rfc822.Message(strfile)` возвращает объект с интерфейсами в виде словарей, что позволяет получить значение заголовка сообщения по строке имени (например, для получения строки заголовка «From» объект индексируется по строке «From»).

Все, что остается в `strfile` после вызова `Message`, является *телом* сообщения, которое здесь не используется, но может быть получено при вызове `strfile.read`. Вызов `Message` читает только часть сообщения, состоящую из заголовков. Заметьте, что `strfile` в действительности является экземпляром стандартного объекта `StringIO.StringIO`. Этот объект заключает исходный текст сообщения (простую строку) в интерфейс типа файла. `rfc822.Message` предполагает интерфейс файла, но не заботится о том, является ли объект настоящим файлом. Еще раз напомним, что *интерфейсы* в Python являются правилами, в соответствии с которыми пишется код, а не конкретными *типами*. Модуль `StringIO` всегда полезен, когда требуется, чтобы строка выглядела как файл.

Я думаю, что сейчас вы уже достаточно хорошо знаете Python, чтобы прочесть этот сценарий и разобраться, как он работает, поэтому вместо лишних слов о его конструкции перейдем к интерактивному сеансу `pymail` и посмотрим на него в действии.

Работа с клиентом командной строки pymail

Запустим `pymail`, чтобы прочесть и удалить e-mail на нашем почтовом сервере и отправить новые сообщения. `Pymail` выполняется на любой машине с Python и сокетами, за-

[1] Подробнее о номерах сообщений в POP будет сказано при рассмотрении PyMailGui далее в этой главе. Интересно, что список номеров удаляемых сообщений не должен сортироваться: номера сохраняют свою действенность на протяжении всего соединения.

Знает кто-нибудь в самом деле, который час?

Маленькое предостережение: простой формат времени, используемый в программе `smtpmail` (и других программах этой книги) не вполне согласуется со стандартом форматирования даты в SMTP. Для большинства серверов это безразлично, и они допускают любой текст даты в строках заголовков даты. Мне не доводилось видеть, чтобы почта не прошла из-за формата даты.

Однако если вы хотите более точно следовать стандарту, то можете форматировать заголовок даты с помощью такого кода (взятого из стандартного модуля `urllib` и допускающего разбор с помощью стандартных средств, например модуля `rfc822` и вызова `time.strptime`):

```
import time
gmt = time.gmtime(time.time())
fmt = '%a, %d %b %Y %H:%M:%S GMT'
str = time.strftime(fmt, gmt)
hdr = 'Date: ' + str
print hdr
```

После выполнения этого кода переменная `hdr` будет выглядеть как

```
Date: Fri, 02 Jun 2000 16:40:41 GMT
```

в отличие от формата даты используемого в программе `smtpmail`:

```
>>> import time
>>> time.ctime(time.time())
'Fri Jun 02 10:23:51 2000'
```

Вызов `time.strftime` позволяет форматировать дату и время произвольным образом (`time.ctime` представляет лишь один стандартный формат), но мы оставим читателю в качестве упражнения возможность поковыряться в действиях всех этих вызовов; обратитесь к описанию модуля `time` в руководстве по библиотеке. Мы также оставим задачу помещения такого кода в файл, допускающий многократное использование, читателям, увлекающимся созданием модулей. Правила для форматирования времени и даты необходимы, но не отличаются изяществом.

гружает почту с любого почтового сервера с интерфейсом POP, на котором у вас есть учетная запись, и отправляет почту через сервер SMTP, указанный в модуле `mailconfig`.

Вот как он действует на моем переносном компьютере под Windows 98; на других машинах он работает идентичным образом. Во-первых, мы запускаем сценарий, вводим пароль POP (помним, что для серверов SMTP пароль не требуется) и ждем, когда появится список почты `pymail`:

```
C:\...\PP2E\Internet\Email>python pymail.py
Password for pop.rmi.net?

[Pymail email client]
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <870f000002f56c39@chevalier>
('+OK 5 messages (7150 octets)', ['1 744', '2 642', '3 4456', '4 697', '5 611'],
 36)
There are 5 mail messages in 7150 bytes
Retrieving: 1 2 3 4 5
1:      676 bytes
```

```
        From=>lumber.jack@TheLarch.com
        Date=>Wed Jul 12 16:03:59 2000
        Subject=>I'm a Lumberjack, and I'm okay
2:      587 bytes
        From=>lutz@rmi.net
        Date=>Wed Jul 12 16:06:12 2000
        Subject=>testing
3:      4307 bytes
        From=>"Mark Hammond" <MarkH@ActiveState.com>
        Date=>Wed, 12 Jul 2000 18:11:58 -0400
        Subject=>[Python-Dev] Python .NET (was Preventing 1.5 extensions...
4:      623 bytes
        From=>Eric.the.Half.a.Bee@semibee.com
        Date=>Wed Jul 12 16:09:21 2000
        Subject=>A B C D E F G
5:      557 bytes
        From=>lutz@rmi.net
        Date=>Wed Jul 12 16:10:55 2000
        Subject=>testing smtpmail
[Press Enter key]
[Pymail] Action? (i, l, d, s, m, q, ?) l 5
-------------------------------------------------------------------------------

Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:45:38 2000)
X-From_: lutz@rmi.net  Wed Jul 12 16:17:58 2000
Return-Path: <lutz@chevalier.rmi.net>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA00415
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:17:57 -0600 (MDT)
Message-Id: <200007122217.QAA00415@chevalier.rmi.net>
From: lutz@rmi.net
To: lutz@rmi.net
Date: Wed Jul 12 16:10:55 2000
Subject: testing smtpmail

Lovely Spam! Wonderful Spam!

-------------------------------------------------------------------------------

[Pymail] Action? (i, l, d, s, m, q, ?) l 4
-------------------------------------------------------------------------------

Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:45:38 2000)
X-From_: Eric.the.Half.a.Bee@semibee.com  Wed Jul 12 16:16:31 2000
Return-Path: <Eric.the.Half.a.Bee@semibee.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA28647
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:16:30 -0600 (MDT)
From: Eric.the.Half.a.Bee@semibee.com
Message-Id: <200007122216.QAA28647@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:09:21 2000
Subject: A B C D E F G

Fiddle de dum, Fiddle de dee,
Eric the half a bee.

-------------------------------------------------------------------------------
```

Как только pymail загрузит электронную почту в список Python на локальной машине клиента, можно водить буквы команд для ее обработки. Команда «l» выводит содержимое сообщения с заданным номером; в данном случае с ее помощью выведены два сообщения, которые мы написали с помощью сценария smtpmail в предыдущем разделе.

Pymail позволяет также получать подсказку по командам, удалять сообщения (фактическое удаление происходит на сервере при выходе из программы) и сохранять сообщения в локальном текстовом файле, имя которого указано в модуле mailconfig:

```
[Pymail] Action? (i, l, d, s, m, q, ?) ?

Available commands:
i     - index display
l n?  - list all messages (or just message n)
d n?  - mark all messages for deletion (or just message n)
s n?  - save all messages to a file (or just message n)
m     - compose and send a new mail message
q     - quit pymail
?     - display this help text

[Pymail] Action? (i, l, d, s, m, q, ?) d 1
[Pymail] Action? (i, l, d, s, m, q, ?) s 4
```

Теперь выберем параметр «m» для *составления* почтового сообщения – pymail просто выполнит сценарий smtpmail, который мы написали в предыдущем разделе, и возобновит свой цикл команд (зачем заново изобретать колесо?). Так как этот сценарий осуществляет отправку с помощью SMTP, можно использовать здесь произвольные адреса «From»; но снова напомним, что обычно не следует этого делать (если, конечно, вы не хотите получить интересные примеры для книги).

Сценарий smtpmail запускается с помощью встроенной функции execfile; если внимательно посмотреть на код pymail, можно заметить, что он передает пустой словарь в качестве пространства имен сценария, чтобы предотвратить конфликт его имен с именами в коде pymail. Функция execfile предоставляет удобный способ повторно использовать имеющийся код, написанный как сценарий верхнего уровня, и поэтому в действительности не предоставляет возможности импортирования. Технически говоря, код в файле *smtplib.py* должен выполниться при импортировании, но только при первом импорте (последующий импорт просто возвращает загруженный объект модуля). Другие сценарии, которые ищут __main__ в атрибуте __name__, обычно вообще не должны выполняться при импорте:

```
[Pymail] Action? (i, l, d, s, m, q, ?) m
From? Cardinal@nice.red.suits.com
To?   lutz@rmi.net
Subj? Among our weapons are these:
Type message text, end with line=(ctrl + D or Z)
Nobody Expects the Spanish Inquisition!
Connecting...
No errors.
Bye.
[Pymail] Action? (i, l, d, s, m, q, ?) q
To be deleted: [1]
Delete?y
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <8e2e0000aff66c39@chevalier>
Deleting messages from server.
Bye.
```

Как уже говорилось, удаление в действительности происходит только при выходе; при завершении pymail командой «q» сценарий сообщает, какие сообщения помещены в очередь на удаление, и просит подтвердить запрос. При подтверждении pymail снова связывается с почтовым сервером и производит вызовы POP для удаления отобранных почтовых сообщений.

Так как pymail загружает почту с сервера в локальный список Python только один раз при начальном запуске, необходимо заново запустить pymail, чтобы снова получить с сервера почту, если нужно посмотреть результат отправки почты и произведенных удалений. В данном случае новое письмо показывается под номером 5, а первоначальное письмо, имевшее номер 1, отсутствует:

```
C:\...\PP2E\Internet\Email>python pymail.py
Password for pop.rmi.net?

[Pymail email client]
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <40310000d5f66c39@chevalier>
...
There are 5 mail messages in 7090 bytes
Retrieving: 1 2 3 4 5
1:      587 bytes
        From=>lutz@rmi.net
        Date=>Wed Jul 12 16:06:12 2000
        Subject=>testing
2:      4307 bytes
        From=>"Mark Hammond" <MarkH@ActiveState.com>
        Date=>Wed, 12 Jul 2000 18:11:58 -0400
        Subject=>[Python-Dev] Python .NET (was Preventing 1.5 extensions...
3:      623 bytes
        From=>Eric.the.Half.a.Bee@semibee.com
        Date=>Wed Jul 12 16:09:21 2000
        Subject=>A B C D E F G
4:      557 bytes
        From=>lutz@rmi.net
        Date=>Wed Jul 12 16:10:55 2000
        Subject=>testing smtpmail
5:      615 bytes
        From=>Cardinal@nice.red.suits.com
        Date=>Wed Jul 12 16:44:58 2000
        Subject=>Among our weapons are these:
[Press Enter key]
[Pymail] Action? (i, l, d, s, m, q, ?) l 5
------------------------------------------------------------------------------

Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:53:24 2000)
X-From_: Cardinal@nice.red.suits.com  Wed Jul 12 16:51:53 2000
Return-Path: <Cardinal@nice.red.suits.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA11127
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:51:52 -0600 (MDT)
From: Cardinal@nice.red.suits.com
Message-Id: <200007122251.QAA11127@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:44:58 2000
Subject: Among our weapons are these:

Nobody Expects the Spanish Inquisition!
```

```
--------------------------------------------------------------------------------
[Pymail] Action? (i, l, d, s, m, q, ?) q
Bye.
```

Наконец, вот содержимое файла с сохраненной почтой, состоящее из одного письма, которое мы попросили сохранить в предыдущем сеансе; оно представляет собой исходный текст сохраненных сообщений со строками-разделителями. Его могут читать как человек, так и машина – в принципе, можно загрузить сохраненную в этом файле почту в список Python в другом сценарии, применив функцию `string.split` к тексту файла и указав строку-разделитель:

```
C:\...\PP2E\Internet\Email>type c:\stuff\etc\savemail.txt

Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:45:38 2000)
X-From_: Eric.the.Half.a.Bee@semibee.com  Wed Jul 12 16:16:31 2000
Return-Path: <Eric.the.Half.a.Bee@semibee.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA28647
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:16:30 -0600 (MDT)
From: Eric.the.Half.a.Bee@semibee.com
Message-Id: <200007122216.QAA28647@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:09:21 2000
Subject: A B C D E F G

Fiddle de dum, Fiddle de dee,
Eric the half a bee.
--------------------------------------------------------------------------------
```

Раскодирование вложений в почтовых сообщениях

В предыдущем разделе мы научились разбирать заголовки и тело почтовых сообщений с помощью модулей `rfc822` и `StringIO`. Однако для некоторых сообщений этого недостаточно. В данном разделе будут представлены инструменты, которые идут дальше, обрабатывая сложную информацию в теле почтового сообщения.

Одна из отрицательных сторон упорной привязанности к почтовому интерфейсу командной строки в telnet связана с тем, что иногда в письма помещаются самые разнообразные данные – картинки, файлы MS Word, архивы в кодировке uuencode, документы в кодировке base64, страницы HTML и даже выполняемые сценарии, при открытии которых можно покалечить компьютер.[1] Не всякое вложение имеет важность, конечно, но в наше время e-mail не всегда состоит только из текста ASCII.

[1] Должен сказать, что я имею в виду почтовые вирусы, появившиеся в 2000 году. Большинство из них связано с тем, что в Microsoft Outlook оказалась «функция», позволявшая почтовым вложениям содержать исполняемые сценарии и позволявшая этим сценариям при открытии и выполнении получать доступ к важным компонентам компьютера. Кроме того, в Outlook была еще одна функция, которая автоматически запускала такие вложенные сценарии при просмотре электронной почты, даже без открытия вложения вручную. Оценить в полной мере такую дыру в защите я предлагаю вам самим, но хочу добавить, что если в какой-либо из почтовых программ этой книги вы будете использовать средства Python для работы с вложениями, то ни в коем случае не выполняйте вложенные программы, за исключением запуска их в ограниченном режиме исполнения Python, о котором рассказывается в главе 15 «Более сложные темы Интернета».

Пока я не преодолел своей привязанности к telnet, мне требовался способ извлечения и обработки этих приложений в командной строке (я попытался просто полностью игнорировать вложения, но долго поступать так было невозможно). К счастью, библиотечные средства Python позволяют обрабатывать такие вложения и стандартные кодировки простым и переносимым образом. Для простоты все последующие сценарии работают с исходным текстом сохраненных почтовых сообщений (или его частями), но их легко можно включить в почтовые программы этой книги, чтобы автоматически извлекать составные части электронного письма.

Раскодирование данных base64

Начнем с чего-нибудь простенького. Почтовые сообщения и вложения часто пересылаются в таких форматах кодировки, как uu или base64; в частности, с помощью одной из этих схем кодировки необходимо кодировать файлы двоичных данных для пересылки в текстовом формате. На приемном конце такие закодированные данные необходимо раскодировать, чтобы просматривать их, открывать или использовать иным способом. Программа Python в примере 11.19 умеет снимать кодировку base64 с данных, хранящихся в файле.

Пример 11.19. PP2E\Internet\Email\decode64.py

```
#!/usr/bin/env python
###############################################################################################
# Раскодирование почтовых вложений, пересылаемых в формате base64. В этой версии
# предполагается, что данные в кодировке base64 помещены в отдельный файл.
# Сценарий не распознает заголовки mime и части. Раскодирование uu осуществляется
# аналогично (uu.decode(iname)), как и раскодирование binhex (binhex.hexbin(iname)).
# То же можно осуществить с помощью модуля mimetools:
# mimetools.decode(input, output, 'base64').
###############################################################################################

import sys, base64

iname = 'part.txt'
oname = 'part.doc'

if len(sys.argv) > 1:
    iname, oname = sys.argv[1:]      # % python prog [iname oname]?

input  = open(iname, 'r')
output = open(oname, 'wb')           # в Windows требуется wb
base64.decode(input, output)         # здесь главная работа
print 'done'
```

Особенно смотреть здесь не на что, поскольку весь процесс трансляции на низком уровне происходит в модуле Python `base64`; мы просто вызываем его метод `decode` с открытыми входным и выходным файлами. Другие схемы кодировки при передаче поддерживаются различными модулями Python – `uu`, `binhex` и т. д. Все они экспортируют интерфейсы, аналогичные `base64`, и просты в обращении; `uu` и `binhex` используют имя выходного файла, находящееся в данных (см. детали в библиотечном руководстве).

Модуль `mimetools` обеспечивает несколько более высокий уровень общности, экспортируя метод `decode`, поддерживающий все схемы кодировки. Требуемый тип раскодирования задается в передаваемом методу аргументе, но конечный результат остается тем же, как показано в примере 11.20.

Пример 11.20. PP2E\Internet\Email\decode64_b.py

```
#!/usr/bin/env python
```

```
#################################################################
# Раскодировать почтовое вложение, переданное в формате base64.
# В этой версии проверяется модуль mimetools.
#################################################################

import sys, mimetools

iname = 'part.txt'
oname = 'part.doc'

if len(sys.argv) > 1:
    iname, oname = sys.argv[1:]               # % python prog [iname oname]?

input  = open(iname, 'r')
output = open(oname, 'wb')
mimetools.decode(input, output, 'base64')     # или 'uuencode', и т. д.
print 'done'
```

Для использования любого из этих сценариев необходимо сначала извлечь данные в кодировке base64 и поместить их в текстовый файл. Сохраните почтовое сообщение в текстовом файле с помощью любого почтового клиента, затем отредактируйте этот файл в любом текстовом редакторе, сохранив только часть в кодировке base64. Наконец, передайте файл данных в сценарий вместе с именем выходного файла, в котором будут сохранены раскодированные данные. Вот пример действия декодировщиков base64 на файл с сохраненными данными; полученный выходной файл оказывается таким же, как тот, который ранее был получен при сохранении вложения в MS Outlook:

```
C:\Stuff\Mark\etc\jobs\test>python ..\decode64.py t4.64 t4.doc
done

C:\Stuff\Mark\etc\jobs\test>fc /B cand.agr10.22.doc t4.doc
Comparing files cand.agr10.22.doc and t4.doc
FC: no differences encountered


C:\Stuff\Mark\etc\jobs\test>python ..\decode64_b.py t4.64 t4.doc
done

C:\Stuff\Mark\etc\jobs\test>fc /B cand.agr10.22.doc t4.doc
Comparing files cand.agr10.22.doc and t4.doc
FC: no differences encountered
```

Извлечение и раскодирование всех частей сообщения

Процедура раскодирования из предыдущего раздела требует много ручных действий и чревата совершением ошибок; кроме того, она обрабатывает только один тип кодировки (base64) и раскодирует только одну составляющую почтового сообщения. С помощью некоторой дополнительной логики можно существенно улучшить ее, используя средства раскодирования многочастевых сообщений из модуля Python `mhlib`. Например, сценарий примера 11.21 умеет извлекать, раскодировать и сохранять все компонеты почтового сообщения за один шаг.

Пример 11.21. PP2E\Internet\Email\decodeAll.py

```
#!/usr/bin/env python
#########################################################################################
# Раскодировать все почтовые вложения, переданные в закодированном формате:base64, uu и т. д.
# Скопируйте целиком сообщение в mailfile и выполните:
#    % python ..\decodeAll.py mailfile
# В итоге создаются один или более файлов mailfile.part*.
#########################################################################################
```

```
import sys, mhlib
from types import *
iname = 'mailmessage.txt'

if len(sys.argv) == 3:
    iname, oname = sys.argv[1:]            # % python prog [iname [oname]?]?
elif len(sys.argv) == 2:
    iname = sys.argv[1]
    oname = iname + '.part'

def writeparts(part, oname):
    global partnum
    content = part.getbody()               # раскодируемое содержимое или список
    if type(content) == ListType:          # несколько частей: рекурсия для каждой
        for subpart in content:
            writeparts(subpart, oname)
    else:                                  # либо одна раскодированная часть
        assert type(content) == StringType # использовать имя файла, если оно есть в заголовках
        print; print part.getparamnames()  # или создать с помощью счетчика
        fmode = 'wb'
        fname = part.getparam('name')
        if not fname:
            fmode = 'w'
            fname = oname + str(partnum)
            if part.gettype() == 'text/plain':
                fname = fname + '.txt'
            elif part.gettype() == 'text/html':
                fname = fname + '.html'
        output = open(fname, fmode)        # в Windows режим должен быть 'wb'
        print 'writing:', output.name      # для файлов doc, а не 'w'
        output.write(content)
        partnum = partnum + 1

partnum = 0
input   = open(iname? 'r')                 # открыть файл письма
message = mhlib.Message('.', 0, input)     # папка и число в аргументах игнорируются
writeparts(message, oname)
print 'done: wrote %s parts' % partnum
```

Поскольку `mhlib` распознает составляющие части почтового сообщения, этот сценарий обрабатывает сообщение целиком: не нужно редактировать его, чтобы вручную извлечь компоненты. Кроме того, компоненты объекта `mhlib.Message` представляют уже раскодированные части почтового сообщения – к тому моменту, как мы получаем их из объекта, все необходимые операции по снятию кодировки uu, base64 и других уже автоматически применены к компонентам письма. Модуль `mhlib` в состоянии автоматически определять тип кодировки и осуществлять раскодирование; он поддерживает одновременно все стандартные схемы кодировки, а не какой-то конкретный тип, например base64.

Чтобы воспользоваться этим сценарием, сохраните исходный текст почтового сообщения в локальном файле (с помощью любой почтовой утилиты) и укажите имя файла в командной строке сценария. Ниже показан вывод при извлечении и раскодировании этим сценарием двух сохраненных файлов почтовых сообщений, *t4.eml* и *t5.eml*:

```
C:\Stuff\Mark\etc\jobs\test>python ..\decodeall.py t4.eml

['charset']
writing: t4.eml.part0.txt
```

```
['charset']
writing: t4.eml.part1.html

['name']
writing: cand.agr10.22.doc
done: wrote 3 parts


C:\Stuff\Mark\etc\jobs\test>python ..\decodeall.py t5.eml

['charset']
writing: t5.eml.part0.txt

['name']
writing: US West Letter.doc
done: wrote 2 parts
```

Конечным результатом раскодирования сообщения является набор из одного или нескольких локальных файлов, содержащих раскодированное содержимое каждой части сообщения. Так как получаемые локальные файлы составляют суть действия этого сценария, он должен присвоить создаваемым файлам осмысленные имена. Данный сценарий применяет следующие правила образования имен:

- Если с составляющей сообщения связан параметр «name», сценарий сохраняет ее байты в локальном файле с соответствующим именем. Обычно при этом используется исходное имя файла на той машине, откуда исходит почта.
- В иных случаях сценарий создает для составляющей уникальное имя файла, добавляя к имени исходного почтового файла суффикс «partN» и пытаясь угадать расширения файла исходя из типа файла составляющей, указанного в сообщении.

Например, сообщение, сохраненное как *t4.eml*, состоит из тела сообщения, альтернативной кодировки HTML-тела сообщения и вложенного файла документа Word. При раскодировании *t4.eml*:

- В первых двух составляющих сообщения нет параметра «name», поэтому сценарий генерирует имена исходя из имени файла и типов составляющих – *t4.eml.part0.txt* и *t4.eml.part1.html* – простой текст и код HTML, соответственно. На большинстве машин при щелчке по выходному файлу HTML он должен открыться в веб-броузере для форматированного просмотра.
- Последнему вложению при создании было дано явное имя *cand.agr10.22.doc*, поэтому оно непосредственно используется в качестве имени выходного файла. Обратите внимание, что при отправке это был файл документа MS Word; если пересылка прошла правильно, двойной щелчок по третьему выходному файлу, созданному этим сценарием, должен открыть его в Word.

В библиотеке Python есть дополнительные средства для раскодирования данных, полученных из Сети, но за дальнейшими подробностями мы отсылаем к руководству по библиотеке. И снова при работе с этим сценарием раскодирования требуется некоторая ручная работа – пользователь должен сохранить файл почты и ввести команду для выделения частей сообщения в отдельные файлы, но его достаточно для обработки многочастевых сообщений, и он работает на любой машине, где есть Python. Кроме того, демонстрируемые интерфейсы раскодирования могут применяться в интерактивных почтовых клиентах более автоматизированным образом.

Например, раскодированный текст составляющей почтового сообщения можно автоматически отправлять обрабатывающим программам (например, броузерам, текстовым редакторам, Word), а не записывать в локальные файлы. Можно также сохранять его во временном локальном файле и открывать автоматически (в Windows вре-

менный файл можно открыть командой DOS *start* из `os.system`). В действительности в популярных почтовых программах типа Outlook с помощью таких схем поддерживается открытие вложений. Написанные на Python почтовые интерфейсы пользователя тоже могут действовать таким способом – намек на направление, в котором продолжится эта глава.

Почтовый клиент PyMailGui

Как завершающую часть рассказа о средствах электронной почты данный раздел представляет *PyMailGui* – программу Python/Tkinter, реализующую интерфейс пользователя электронной почты на стороне клиента. Она представлена с двух сторон – как пример сценария Python для Интернета и как пример, связывающий вместе другие средства, с которыми мы уже знакомы, например потоки и GUI Tkinter.

Как и написанная ранее программа `pymail`, PyMailGui целиком выполняется на локальном компьютере. Электронная почта загружается с удаленных почтовых серверов и отправляется на них через сокеты, но программа и ее интерфейс пользователя выполняются локально. По этой причине PyMailGui называется *клиентом* электронной почты: он реализует инструменты Python для стороны клиента, чтобы вести диалог с почтовыми серверами с локальной машины. В действительности в некоторых отношениях PyMailGui является развитием `pymail` путем добавления GUI. Однако в отличие от `pymail`, PyMailGui предоставляет интерфейс пользователя с достаточно полным набором функций: операции с электронной почтой осуществляются с помощью действий «point-and-click» («укажи-и-щелкни»).

Зачем нужен PyMailGui?

Как и многие примеры, представленные в этой книге, PyMailGui является также практической, полезной программой. В действительности я запускаю его на самых разных машинах, чтобы проверить свою электронную почту, путешествуя по свету и преподавая Python (это способ справиться с провайдерами Интернета, отказывающими в telnet). Вряд ли PyMailGui затмит в ближайшее время Microsoft Outlook, но мне он нравится по двум причинам:

Он переносим

PyMailGui выполняется на любой машине, где установлены сокеты и Python с Tkinter. Поскольку электронная почта пересылается с помощью библиотек Python, пригодно любое соединение с Интернетом. Более того, поскольку интерфейс пользователя кодируется с помощью расширения Tkinter, PyMailGui будет в неизменном виде действовать под Windows, в системе X Windows (Unix, Linux) и на Macintosh.

Пакет Microsoft Outlook более богат функциями, но выполняться он должен под Windows, а точнее, на одной и той же машине с Windows. Поскольку он обычно удаляет загруженную электронную почту с сервера и сохраняет ее на машине клиента, нельзя выполнять Outlook на нескольких машинах, не посылая почту на все эти машины. Напротив, PyMailGui сохраняет и удаляет почту только по требованию, а потому несколько более любезен по отношению к тем, кто проверяет свою почту по случаю на произвольных машинах.

Он программируем

PyMailGui можно превратить во все что угодно, поскольку он полностью программируем. В действительности это по-настоящему замечательная черта PyMailGui и

программного обеспечения с открытым кодом типа Python в целом – так как исходный код PyMailGui полностью доступен, вы полностью управляете его дальнейшим развитием. С закрытыми коммерческими продуктами типа Outlook такой степени контроля у вас и близко не бывает; обычно вы получаете то, что по мнению большой компании вам нужно, наряду с теми ошибками, которые могут присутствовать по вине этой компании.

Будучи сценарием Python, PyMailGui является значительно более гибким инструментом. Например, я могу быстро изменить структуру интерфейса, отключить функции или добавить новые, внеся изменения в исходный код Python. Не нравится, как выводится список почтовых сообщений? Поменяйте несколько строк кода и настройте его, как вам хочется. Хотите автоматически сохранять и удалять почту при загрузке? Добавьте еще немного кода и кнопки. Надоело видеть рекламную почту? Добавьте несколько строк кода, обрабатывающего текст, чтобы загрузить функцию, отфильтровывающую спам. Это лишь некоторые примеры. Смысл в том, что поскольку PyMailGui написан на языке сценариев высокого уровня, который легко сопровождать, такие настройки осуществляются относительно просто и могут даже доставить массу удовольствия.[1]

Стоит также упомянуть, что PyMailGui достигает такой переносимости и управляемости, реализуя при этом полноценный почтовый интерфейс, примерно в 500 строках программного кода. В нем нет такого обильного украшательства, как в коммерческих продуктах, но тот факт, что ему удается настолько приблизиться к ним при таком малом количестве строк кода, свидетельствует о мощи языка Python и его библиотек.

Запуск PyMailGui

Конечно, чтобы самостоятельно управлять PyMailGui, необходимо иметь возможность его запускать. Для PyMailGui требуется лишь компьютер с каким-либо видом подключения к Интернету (достаточно PC с доступом к Интернету по телефонной линии и модема) и установленным на нем Python с включенным расширением Tkinter. Перенос Python на Windows его содержит, поэтому пользователи Windows-компьютеров имеют возможность сразу запустить эту программу, щелкнув по ее значку (самоинсталлирующийся перенос под Windows находится на прилагаемом к книге CD и на *http://www.python.org*). Необходимо также изменить файл *mailconfig.py* в каталоге примеров e-mail, чтобы отразить в нем параметры вашей учетной записи; подробнее об этом – по ходу диалога с системой.

Стратегия представления

Вполне может оказаться, что PyMailGui является самой длинной программой в этой книге (его основной сценарий состоит примерно из 500 строк, считая пустые строки и комментарии), но новых библиотечных интерфейсов сверх тех, которые уже были показаны, в нем немного. Например:

- Интерфейс PyMailGui строится с помощью расширения Python Tkinter с использованием уже знакомых окон списков, кнопок и текстовых окон.
- Для выделения из сообщений заголовков и текста применяется модуль Python `rfc822` анализа почтовых заголовков.

---

[1] Пример: я добавил код, извлекающий пароль POP из локального файла вместо получения из всплывающего окна, за 10 минут, и он занял меньше 10 строк. Конечно я неплохо программирую, но все равно ждать появления новых функций в Outlook пришлось бы значительно дольше.

- Для получения, отправки и удаления почтовых сообщений через сокеты используются библиотечные модули Python POP и SMTP.
- Если в вашем интерпретаторе Python установлены потоки выполнения, они применяются с целью избежать блокировки при длительных операциях с почтой (загрузке, отправке, удалении).

Для просмотра и составления сообщений мы будем повторно использовать объект `TextEditor`, который написали в главе 9 «Более крупные примеры GUI», для загрузки и удаления почты с сервера применим простые средства модуля `pymail`, написанного в начале этой главы, а получение параметров электронной почты осуществим с помощью модуля `mailconfig` из данной главы. PyMailGui в значительной мере представляет собой упражнение в комбинировании уже имеющихся инструментов.

С другой стороны, ввиду большой длины программы мы не станем исчерпывающим образом документировать ее код. Вместо этого мы начнем с описания работы PyMailGui с точки зрения конечного пользователя. После этого мы перечислим новые модули исходного кода системы без каких-либо комментариев, чтобы в дальнейшем заняться их изучением.

Как и для большинства длинных конкретных примеров этой книги, в данном разделе предполагается, что читатель достаточно хорошо знает Python, чтобы самостоятельно разобраться в коде. Если вы читали книгу последовательно, то должны также быть достаточно знакомы с Tkinter, потоками и почтовыми интерфейсами, чтобы понять библиотечные средства, которые здесь применяются. Если возникнут сложности, может потребоваться освежить в памяти эти темы, излагаемые в более ранних главах.

Взаимодействие с PyMailGui

Чтобы легче было понять этот практический пример, рассмотрим вначале фактически выполняемые PyMailGui операции – взаимодействие с пользователем и функции обработки почты – и только потом обратимся к коду Python, реализующему их. При чтении этого материала не стесняйтесь заглянуть вперед в листинги кода, которые следуют за снимками экранов, но обязательно прочтите также этот раздел, в котором я объясняю все тонкости конструкции PyMailGui. После этого раздела вы можете самостоятельно изучать листинги исходного кода Python, дающие лучшее и более полное объяснение, чем то, которое можно написать на естественном языке.

Запуск

PyMailGui является программой Python/Tkinter, запускаемой при выполнении файла сценария верхнего уровня *PyMailGui.py*. Как и другие программы Python, PyMailGui можно запускать из системной командной строки, щелчком по значку с именем его файла в окне менеджера файлов или нажатием кнопки в панелях запуска PyDemos или PyGadgets. Независимо от способа запуска первоначально PyMailGui выводит окно, показанное на рис. 11.10.

Это главное окно PyMailGui – отсюда начинаются все операции. В его состав входят:

- Кнопка помощи (светло-голубая полоса вверху)
- Область для списка полученных почтовых сообщений (средняя белая часть), допускающая щелчки мышью
- Нижняя панель с кнопками для обработки сообщений, выделенных в области списка

При обычной работе пользователь загружает свою почту, выбирает щелчком сообщение в области списка и нажимает одну из нижних кнопок для его обработки. Первона-

Программное обеспечение с открытым исходным кодом и Camaro

Приведем аналогию, которая подчеркнет важность такого свойства PyMailGui, как управляемость. Кое-кто еще помнит времена, когда считалось совершенно нормальным, если владелец автомобиля переделывал или чинил его. Я с удовольствием вспоминаю, как в годы моей юности мы с друзьями грудились под капотом Camaro 1970 года выпуска, ковыряясь в двигателе и переделывая его. Немного потрудившись, мы могли сделать его быстрым, порывистым и громким по своему вкусу. Кроме того, поломка в какой-либо из этих старых машин не обязательно означала конец света. По крайней мере, был шанс, что я заставлю это машину ехать, не прибегая к чьей-либо помощи.

Сегодня это уже не так. С появлением электронного управления и забитого битком отсека двигателя автовладельцу обычно лучше всего вернуть машину продавцу или обратиться к профессиональному механику во всех случаях, кроме простейшего ремонта. В целом автомобиль перестал быть изделием, ремонтировать которое может сам пользователь. И если произойдет поломка в моем новеньком блестящем джипе, я, скорее всего, прочно застряну, пока сертифицированный ремонтник не возьмет меня на буксир или займется починкой.

Мне нравится представлять модели закрытого и открытого программного обеспечения в том же духе. При пользовании Microsoft Outlook я привязан как к набору функций, диктуемому крупной фирмой, так и к ошибкам, которые могут таиться в продукте. Но пользуясь таким программируемым средством, как PyMailGui, я имею возможность залезть под капот. Я могу добавлять функции, настраивать систему по своему вкусу и самостоятельно избавляться от обнаруженных ошибок. И делать это я могу значительно быстрее, чем появится очередной патч или выпуск Outlook.

В конечном счете, программное обеспечение с открытым кодом дает свободу. Последнее слово остается за пользователем, а не какой-то далекой компанией. Конечно, не всякий захочет возиться со своей машиной. С другой стороны, отказы в программах случаются значительно чаще, чем в машинах, а написание сценариев на Python – существенно менее грязная работа, чем автомеханика.

чально не выводится никаких сообщений: необходимо загрузить их, как будет показано чуть ниже. Однако прежде чем сделать это, нажмем на голубую полосу вверху и посмотрим имеющуюся подсказку. На рис. 11.11 показано окно подсказки, которое при этом появляется.

Основную часть этого окна составляет просто кусок текста в текстовом окне с прокруткой, а также две кнопки внизу. Весь текст подсказки представлен в программе

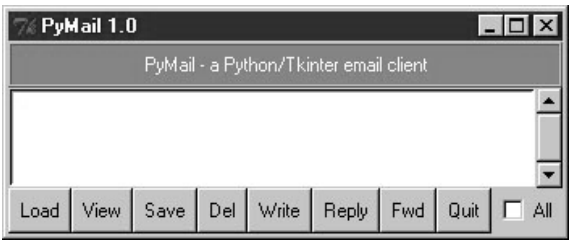


Рис. 11.10. Запуск главного окна PyMailGui

Python как одна строка в тройных кавычках. Можно было бы навести красоту и вызвать веб-броузер для просмотра подсказки в формате HTML, но вполне достаточно и простого текста.[1] Кнопка Cancel закрывает это немодальное (то есть неблокирующее) окно. Более интересно, что кнопка Source открывает окно для просмотра исходного кода главного сценария PyMailGui, показанное на рис. 11.12.

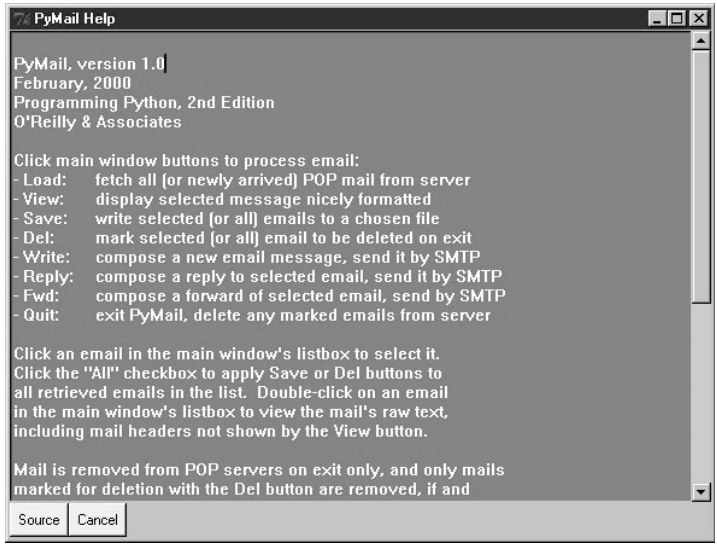
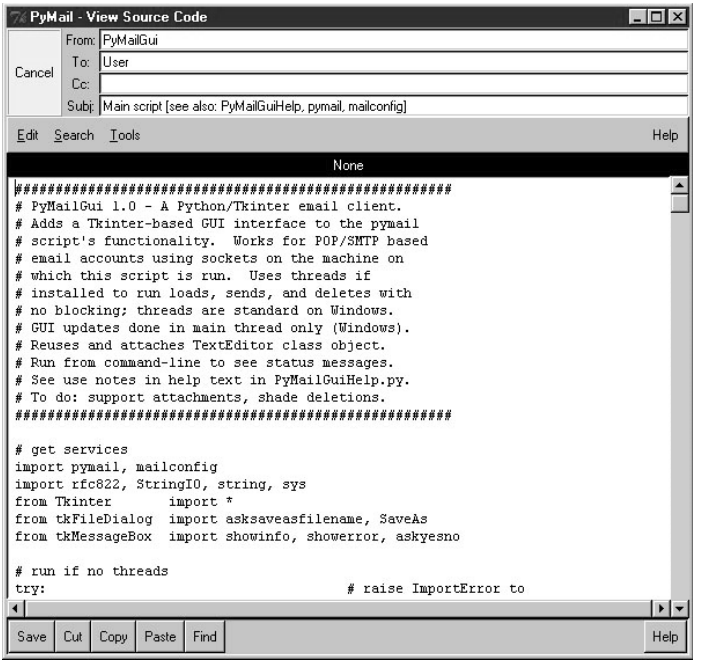


Рис. 11.11. Всплывающее окно подсказки PyMailGui

Не всякая программа покажет вам свой исходный код, но PyMailGui следует лейтмотиву открытого исходного кода Python. Если оставить в стороне политику, то основной интерес здесь представляет то, что для просмотра исходного кода PyMailGui использует то же окно, что и для просмотра почты. Вся информация поступает изнутри PyMailGui; но то же самое окно используется для отображения и редактирования почты, передаваемой через Сеть, поэтому рассмотрим сейчас его формат:

- Верхняя часть состоит из кнопки Cancel, служащей для удаления этого немодального окна, а также раздела для отображения строк заголовка сообщения – «From:», «To:» и т. д.
- Основу этого окна составляет повторное использование объекта класса `TextEditor`, который был написан ранее для программы PyEdit, – PyMailGui просто вставляет экземпляр `TextEditor` в каждое окно для просмотра и написания сообщения, бесплатно получая компонент полнофункционального текстового редактора. В действительности все в этом окне, кроме кнопки Cancel и строк заголовков, реализовано с помощью `TextEditor`, а не PyMailGui.

[1] В действительности вывод подсказки был вначале еще менее красивым: первоначально текст выводился в стандартном информационном окне, генерируемом вызовом Tkinter `showinfo`, использовавшимся ранее в этой книге. Под Windows все было хорошо (по крайней мере, при небольшом объеме текста), но в Linux происходила неудача из-за ограничений по умолчанию на длину строки в информационном окне – строки переносились так плохо, что становились нечитаемыми. Мораль: если хотите пользоваться `showinfo` и при том под Linux, делайте строчки короткими и объем текста маленьким.




```
##########################################################
# PyMailGui 1.0 - A Python/Tkinter email client.
# Adds a Tkinter-based GUI interface to the pymail
# script's functionality.  Works for POP/SMTP based
# email accounts using sockets on the machine on
# which this script is run.  Uses threads if
# installed to run loads, sends, and deletes with
# no blocking; threads are standard on Windows.
# GUI updates done in main thread only (Windows).
# Reuses and attaches TextEditor class object.
# Run from command-line to see status messages.
# See use notes in help text in PyMailGuiHelp.py.
# To do: support attachments, shade deletions.
##########################################################

# get services
import pymail, mailconfig
import rfc822, StringIO, string, sys
from Tkinter       import *
from tkFileDialog  import asksaveasfilename, SaveAs
from tkMessageBox  import showinfo, showerror, askyesno

# run if no threads
try:                                    # raise ImportError to
```

Рис. 11.12. Окно просмотра исходного кода PyMailGui

Например, при выборе меню Tools в текстовой части окна и пункта Info в нем мы получаем стандартное окно объекта `TextEditor` из PyEdit для статистики файла – в точности то же всплывающее окно, что и в автономном текстовом редакторе PyEdit и программе просмотра графики PyView из главы 9 (рис. 11.13).

В действительности это уже *третье* повторное использование `TextEditor` в данной книге: PyEdit, PyView и теперь PyMaiGui предоставляют пользователям один и тот же интерфейс редактирования текста, поскольку все они используют один и тот же объект `TextEditor`. Решая задачу показа исходного кода, можно было бы просто запустить программу PyEdit, передав ей в командной строке имя файла с исходным текстом (подробности в описании PyEdit ранее в этой книге). PyMailGui вместо этого прикрепляет экземпляр.

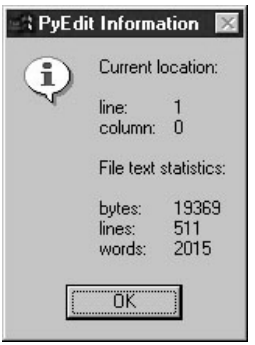


Рис. 11.13. Информационное окно TextEditor, прикрепленное к PyMailGui

Для отображения электронной почты PyMailGui вставляет ее текст в прикрепленный объект `TextEditor`; чтобы написать письмо, PyMailGui показывает `TextEditor` и потом берет из него текст для передачи через Сеть. Помимо очевидной простоты при таком повторном использовании кода упрощается распространение усовершенствований и исправлений – все изменения объекта `TextEditor` автоматически наследуются PyMailGui, PyView и PyEdit.

Загрузка почты

Теперь вернемся к главному окну PyMailGui и щелкнем по кнопке Load, чтобы получить входящую почту по протоколу POP. Как и `pymail`, функция загрузки PyMailGui получает параметры учетной записи из модуля `mailconfig`, текст которого приведен в примере 11.15, поэтому проверьте, чтобы в этом файле были отражены параметры вашей электронной почты (то есть имена серверов и имена пользователей), если конечно хотите читать с помощью PyMailGui свою почту.

О параметре учетной записи *password* следует сказать дополнительно. Есть два места, откуда он может поступать в PyMailGui:

Локальный файл

Если поместить имя локального файла, содержащего пароль, в модуль `mailconfig`, PyMailGui при необходимости загрузит пароль из этого файла.

Всплывающий диалог

Если не помещать в `mailconfig` имя файла с паролем (или PyMailGui не может по каким-то причинам получить пароль из этого файла), PyMailGui запросит пароль, когда он потребуется.

Рис. 11.14 показывает приглашение для ввода пароля, которое выводится, если пароль не записан в локальном файле. Обратите внимание, что вводимый пароль не отображается – параметр `show='*'` поля `Entry` в этом окне указывает Tkinter, что вводимые символы должны отображаться как звездочки (этот параметр аналогичен по духу модулю ввода с консоли `getpass`, с которым мы встречались в этой главе, и параметру HTML `type=password`, с которым мы встретимся позже). Введенный пароль находится только в памяти вашей машины; PyMailGui никуда не записывает его для постоянного хранения.

Обратите также внимание, что параметр локального файла с паролем требует, чтобы пароль хранился в незашифрованном виде в файле на локальном компьютере клиента. Это удобно (не нужно каждый раз заново вводить пароль для получения почты), но не очень хорошо, если машина совместно используется несколькими пользователями; оставьте этот параметр пустым в `mailconfig`, если предпочитаете всегда вводить пароль в диалоговом окне.

Рис. 11.14. Окно для ввода пароля PyMailGui

Получив настройки почты и тем или иным способом пароль, PyMailGui постарается загрузить всю входящую почту с сервера POP. PyMailGui повторно использует средства загрузки почты из модуля `pymail`, приведенного в примере 11.18, который в свою очередь использует для получения почты стандартный модуль Python `poplib`.

Выполнение длительных операций пересылки почты в потоках

Однако в конечном итоге функция загрузки должна пересылать данные через сокеты. Если у вас, как и у меня, большой объем переписки, ее работа может потребовать значительного времени. Чтобы не блокировать GUI во время загрузки, PyMailGui порождает поток, в котором операция получения почты с сервера происходит параллельно остальной части программы. Основная программа продолжает реагировать на события окон (то есть перерисовывает отображение после того, как оно было скрыто под другим окном) во время загрузки электронной почты. Узнать о том, что идет загрузка почты в фоновом режиме, можно по всплывающему диалоговому окну, выводимому PyMailGui, которое показано на рис. 11.15.

Рис. 11.15. Окно ожидания при загрузке почты PyMailGui (выполняемой в отдельном потоке)

Этот диалог захватывает фокус и потому отключает остальные кнопки GUI на время загрузки. Он остается наверху на протяжении всей загрузки и автоматически закрывается, когда заканчивается загрузка. Аналогичные всплывающие окна появляются во время других длительных операций с сокетами (*отправки* и *удаления* электронной почты), но сам GUI сохраняет жизнеспособность, поскольку эти операции выполняются в потоках.

В системах, где нет потоков, PyMailGui во время таких длительных операций переходит в заблокированное состояние (вместо операции порождения потока ставится заглушка, выполняющая простой вызов функции). Поскольку GUI без потоков оказывается фактически мертв, на таких платформах при закрытии и открытии GUI во время загрузки почты его содержимое стирается или каким-то образом искажается.[1] По умолчанию потоки включены на большинстве платформ, где выполняется Python (включая Windows), поэтому вряд ли вы увидите такие странности на своей машине.

Одно замечание: как отмечалось при изучении GUI FTP ранее в этой главе, в MS Windows обрабатывать окно может только тот поток, который его создал. Из-за этого в PyMailGui внутри потоков, загружающих, отправляющих и удаляющих электронную почту, не делается ничего, относящегося к интерфейсу пользователя. Вместо этого основная программа продолжает отвечать на события и обновления интерфейса пользователя и ждет, пока потоки пересылки сообщений не выставят глобальный флаг «я закончил». Вспомните, что потоки совместно используют глобальную (то есть принадлежащую модулю) память; так как в PyMailGui одновременно может быть не более двух активных потоков – основной программы и пересылки почты, – одного глобального флага достаточно для реализации механизма связи между потоками.

---

[1] Если хотите посмотреть, как это происходит, измените код PyMailGui так, чтобы класс `fakeThread` в начале файла *PyMailGui.py* был всегда определен (по умолчанию он создается только при невозможности импорта модуля `thread`), и попробуйте накрыть и открыть главное окно во время операции загрузки, отправки или удаления. Окно не будет перерисовываться, потому что PyMailGui с одним потоком занят работой через сокет.

Интерфейс загрузки с сервера

Поскольку операция загрузки в действительности является операцией с сокетами, PyMailGui автоматически соединится с вашим почтовым сервером, используя те возможности соединения, которые есть на машине, где он выполняется. Например, если соединение с Сетью осуществляется через модем и в данный момент соединения нет, Windows автоматически выведет стандартный диалог соединения. Рис. 11.16 показывает тот диалог, который я вижу на своем переносном компьютере. Если PyMailGui выполняется на машине с выделенным соединением с Интернетом, используется последнее.

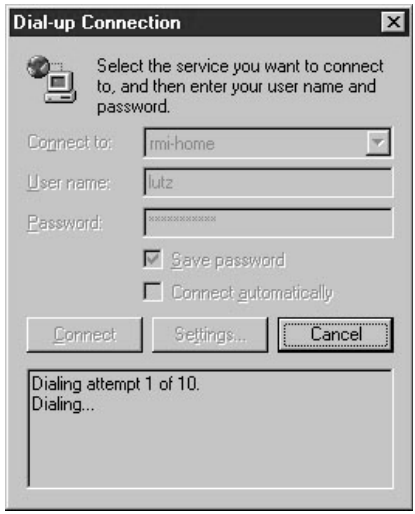


Рис. 11.16. Диалоговое окно соединения PyMailGui (в Windows)

Когда PyMailGui завершает загрузку электронной почты, он заполняет окно списка главного окна сообщениями с почтового сервера и прокручивает его до показа последнего полученного. На рис. 11.17 показано, как выглядит главное окно на моей машине.

Технически кнопка Load при первом своем нажатии загружает всю почту, а при последующих нажатиях – только вновь поступившие сообщения. PyMailGui следит за

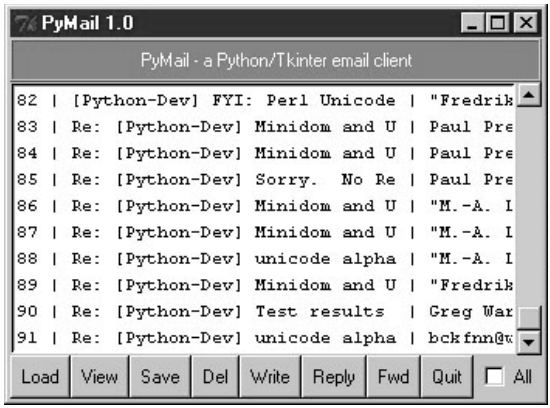


Рис. 11.17. Главное окно PyMailGui после загрузки

тем, какое сообщение было загружено последним, и при последующих загрузках запрашивает только сообщения с номерами, большими последнего. Загруженная почта хранится в памяти в списке Python, чтобы не загружать ее каждый раз снова. Как и в показанном ранее простом интерфейсе командной строки `pymail`, PyMailGui не удаляет почту с сервера при загрузке. Если вы действительно не хотите видеть какое-то письмо при загрузке в будущем, то должны явным образом удалить его (подробнее об этом далее).

Как и в большинстве GUI этой книги, размеры главного окна можно изменять. На рис. 11.18 показано, как будет выглядеть главное окно, если растянуть его так, чтобы показались дополнительные элементы электронной почты. В записях в главном списке показано лишь то, что дает пользователю представление о содержании письма – в каждой записи конкатенированы части строк заголовков сообщения «Subject:», «From:» и «Date:», разделяемые символом | и с проставлением впереди номера сообщения согласно POP (например, в данном списке имеется 91 сообщение). Колонки не всегда точно выровнены (одни заголовки короче, чем другие), но этого достаточно, чтобы указать на содержание сообщения.

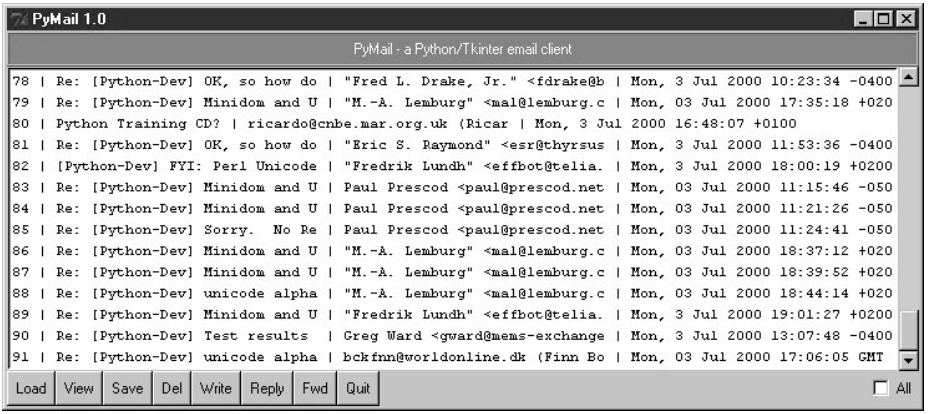


Рис. 11.18. Главное окно PyMailGui с измененными размерами

Как мы уже видели, при загрузке почты с сервера выполняется много действий – клиент (машина, на которой выполняется PyMailGui) должен соединиться с сервером (машиной, где находится ваша почтовая учетная запись) через сокет и передать байты через соединения Интернета. Если что-то пойдет не так, PyMailGui выведет стандартное диалоговое окно ошибки и сообщит, что произошло. Например, если PyMailGui вообще не сможет установить соединение, вы получите окно, вроде показанного на рис. 11.19.

Рис. 11.19. Окно сообщения PyMailGui об ошибке соединения

Здесь выводятся лишь тип и данные исключительной ситуации Python. Если указаны неправильное имя пользователя или пароль (в модуле `mailconfig` или окне для ввода пароля), появится сообщение, показанное на рис. 11.20.

Рис. 11.20. Окно сообщения PyMailGui о неверном пароле

Это окно показывает исключительную ситуацию, возбужденную модулем Python `poplib`. Если PyMailGui не может связаться с сервером (например, он не работает или неверно указано его имя в `mailconfig`), выводится окно, показанное на рис. 11.21.

Рис. 11.21. Окно сообщения PyMailGui о неправильно указанном или неработающем сервере

Отправка почты

После загрузки электронной почты можно обрабатывать сообщения с помощью кнопок главного окна. Можно, однако, в любой момент отправить новые сообщения, даже до загрузки почты. При нажатии в главном окне кнопки Write создается окно, в котором можно составить новое письмо; оно показано на рис. 11.22.

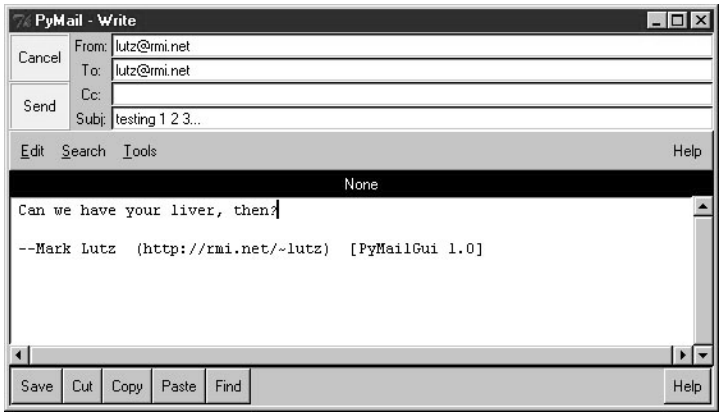


Рис. 11.22. Окно PyMailGui для составления сообщения

Это такое же окно, как для просмотра исходного кода, которое мы видели чуть выше, – в нем есть поля для ввода элементов строк заголовков и встроенный объект `TextEditor`, в котором можно написать тело нового сообщения. В операциях создания сообщений PyMailGui автоматически заполняет строку «From» и вставляет строку подписи («–Mark...»)согласно установкам в модуле `mailconfig`. Можете заменить их любым текстом, но значения по умолчанию автоматически берутся из `mailconfig`.[1]

Здесь также появляется новая кнопка «Send»: при ее нажатии текст, введенный в тело этого окна, пересылается по адресам, указанным в строках «To:» и «Cc:» с помощью модуля Python `smtplib`. PyMailGui помещает заголовки, которые вы ввели в полях, в качестве строк почтовых заголовков в отправляемом письме. Для отправки по нескольким адресам следует перечислить их, разделяя символом «;», в строках «To:» и «Cc:» (ниже будет приведен пример). В данном письме я указал в заголовке «To:» свой собственный почтовый адрес, чтобы в иллюстративных целях отправить письмо самому себе.

Как мы видели, `smtplib` в конечном итоге пересылает байты серверу через сокет. Это может оказаться длительной операцией, поэтому PyMailGui также передает ее выполнение порожденному потоку. Во время выполнения потока отправки показывается окно, приведенное на рис. 11.23, и GUI в целом остается жизнеспособным: события перерисовки и перемещения обрабатываются в главном потоке программы, в то время как поток отправки работает с сервером SMTP.

Рис. 11.23. Окно ожидания при отправке почты PyMailGui (выполняемой в отдельном потоке)

Если по какой-либо причине Python не сможет отправить письмо по одному из указанных адресов получателей, будет выведено окно сообщения об ошибке. Если окно ошибки не показано, значит, все сработало правильно и ваше письмо появится в почтовых ящиках получателей на их почтовых серверах. Так как приведенное выше сообщение я послал себе самому, оно покажется в моем ящике при следующем нажатии в главном окне кнопки Load, как можно видеть на рис. 11.24.

Если взглянуть на последний снимок главного окна, то можно заметить, что теперь в нем только два новых письма – с номерами 92 (от Python-Help) и 93 (только что написанное). PyMailGui достаточно сообразителен, чтобы загрузить только два новых письма и поместить их в конец списка загруженной почты.

---

[1] Как и в более ранних примерах этой главы, PyMailGui отправляет почту по SMTP, поэтому обычно можно использовать в строке «From:» произвольный адрес: большинство серверов не проверяет его верности, если только он удовлетворяет формату почтовых адресов в целом. Но еще раз напомню, что этого делать не следует, – пожалуйста, поместите свой собственный адрес электронной почты в ваш экземпляр *mailconfig.py* или введите его в поле «From:» в окне составления письма.

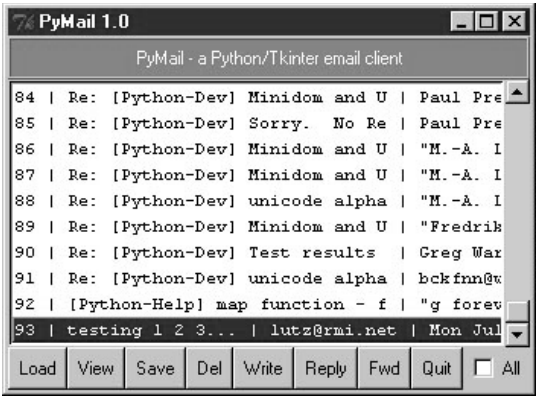


Рис. 11.24. Главное окно PyMailGui после передачи и загрузки

Просмотр электронной почты

Теперь просмотрим почтовое сообщение, которое было отправлено и получено. PyMailGui позволяет просматривать почту в форматированном или исходном виде. Сначала выделим (щелчком) в главном окне сообщение, которое желательно просмотреть, а затем нажмем кнопку View. Появится окно форматированного просмотра сообщения, как на рис. 11.25.

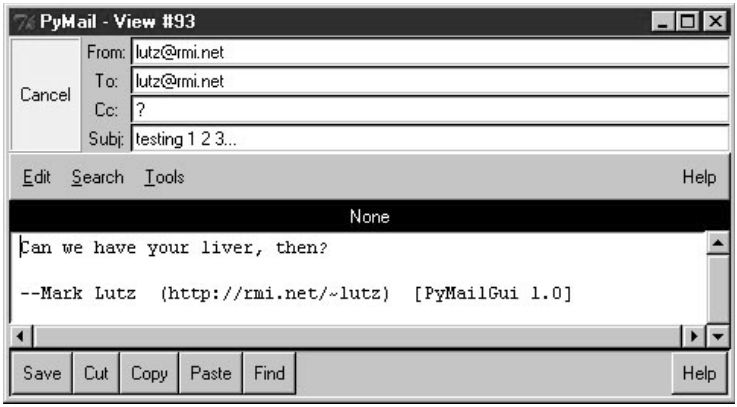


Рис. 11.25. Окно PyMailGui просмотра поступившего сообщения

Это то же самое окно, в котором мы видели исходный код, только теперь все поля заполнены данными, извлеченными из выбранного почтового сообщения. Для выделения строк заголовков из исходного почтового сообщения применяется модуль Python `rfc822`; их текст помещается в правой верхней части окна. После разбора заголовков остается текст тела сообщения (строка, заключенная в объект типа файла `StringIO`), который прочитывается и вставляется для отображения в новый объект `TextEditor` (белый участок в середине окна).

Помимо окна просмотра с красивым форматированием PyMailGui позволяет увидеть необработанный текст почтового сообщения. При двойном щелчке в главном окне по записи сообщения показывается простой неформатированный текст сообщения. Исходный текст сообщения, которое я послал сам себе, показан на рис. 11.26.

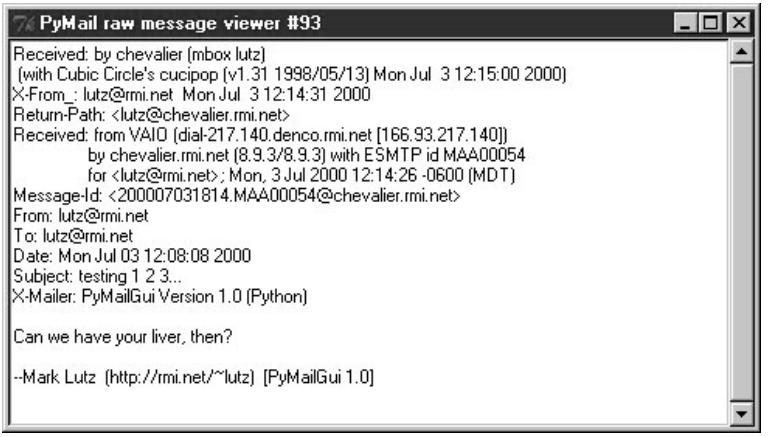


Рис. 11.26. Окно PyMailGui для просмотра исходного текста почтового сообщения

Такое отображение исходного текста может оказаться полезным для просмотра специальных почтовых заголовков, не показываемых в окне форматированного просмотра. Например, необязательный заголовок «X-Mailer:» в исходном тексте идентифицирует программу, передавшую сообщение. PyMailGui автоматически добавляет его в сообщение наряду со стандартными заголовками типа «From:» и «To:». Другие заголовки добавляются в процессе передачи сообщения: заголовки «Received:» указывают имена машин, через которые прошло сообщение на пути к нашему почтовому серверу.

В действительности исходный текст представляет все, что относится к почтовому сообщению – это то, что передается с машины на машину, когда письмо отправлено. Красиво отформатированное отображение просто выделяет участки исходного текста сообщения с помощью стандартных средств Python и помещает их в соответствующие поля экрана на рис. 11.25.

Ответ на сообщения электронной почты и переадресация

PyMailGui позволяет не только читать почту и создавать новую, но и переадресовывать входящую почту и составлять ответ на нее. Чтобы ответить на электронное письмо, выделите его запись в списке главного окна и щелкните по кнопке Reply. Если я отвечу на письмо, которое только что послал самому себе (отдает нарциссизмом, но служит целям демонстрации), появится окно составления письма, показанное на рис. 11.27.

Формат этого окна идентичен тому, который мы видели для операции «Write», за исключением того, что некоторые части автоматически заполняются PyMailGui:

- Строка «From:» устанавливается в соответствии с вашим почтовым адресом в модуле `mailconfig`.
- Строка «To:» инициализируется адресом «From:» исходного сообщения (мы ведь отвечаем тому, кто послал сообщение). Смотрите во врезке «Об обратных адресах» дополнительные сведения об адресах получателей.
- Строка «Subject:» устанавливается равной строке темы исходного сообщения со вставкой впереди «Re: » – стандартный формат продолжения темы.
- В тело сообщения помещаются строка подписи из `mailconfig`, а также текст исходного почтового сообщения. Текст исходного сообщения цитируется с помощью символов `>`, и перед ним помещается несколько строк заголовка исходного сообщения, чтобы дать представление о контексте.

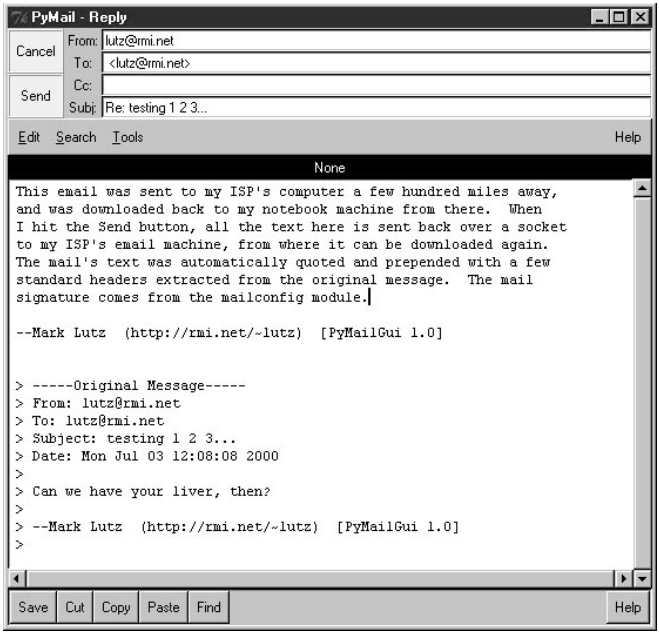


Рис. 11.27. Окно PyMailGui составления ответа

К счастью, все это реализуется значительно проще, чем может показаться. С помощью стандартного модуля Python `rfc822` извлекаются строки заголовков исходного сообщения, а все действия по добавлению в тело исходного сообщения символов цитирования `>` выполняет вызов `string.replace`. Я просто ввожу с клавиатуры все, что хочу сказать в ответ (первый абзац в текстовой области письма), и нажимаю кнопку Send, чтобы снова отправить ответное сообщение в почтовый ящик на моем почтовом сервере. Физически отправка ответа происходит точно так же, как отправка нового сообщения – почта направляется на сервер SMTP в порожденном потоке отправки почты, и появляется окно ожидания конца отправки.

Переадресация сообщения аналогична ответу: выделите сообщение в главном окне, нажмите кнопку «Fwd» и заполните поля и текстовую область появившегося окна составления сообщения. Рис. 11.28 показывает окно, создаваемое для переадресации письма, которое мы ранее написали и получили.

Как и при ответе, «From:» заполняется из `mailconfig`, первоначальный текст в теле сообщения автоматически цитируется, а строка темы устанавливается как тема исходного сообщения со вставкой впереди «Fwd:». Однако строку «To:» я должен заполнить вручную, поскольку это не прямой ответ (письмо не обязательно посылается первоначальному отправителю). Обратите внимание, что я пересылаю письмо в два разных адреса; адреса получателей в полях заголовков «To:» и «Cc:» разделяются символом «;». Кнопка Send в этом окне отправляет переадресуемое письмо по всем адресам, перечисленным в «To:» и «Cc:».

Хорошо, теперь я написал новое письмо, ответил на него и переадресовал его. Ответ и переадресация были посланы также на мой почтовый адрес; если снова нажать на кнопку Load в главном окне, ответное и переадресованное письма должны показаться в списке главного окна. На рис. 11.29 они показаны как сообщения 94 и 95.

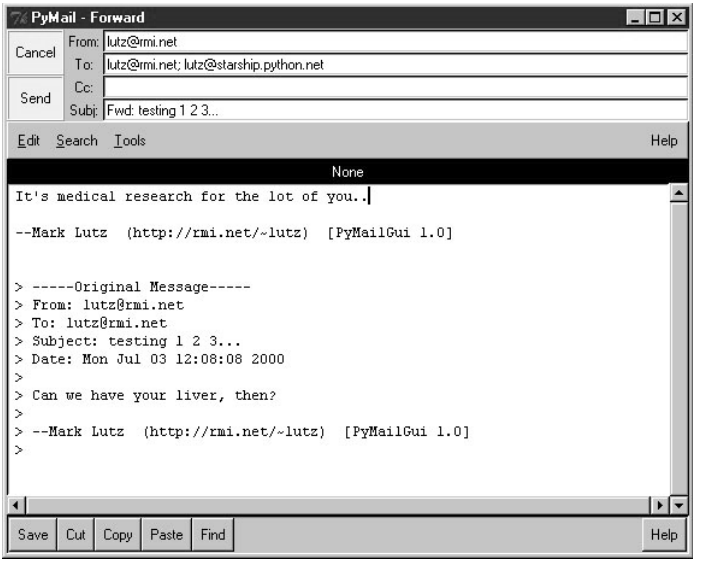


Рис. 11.28. Окно PyMailGui для переадресации письма

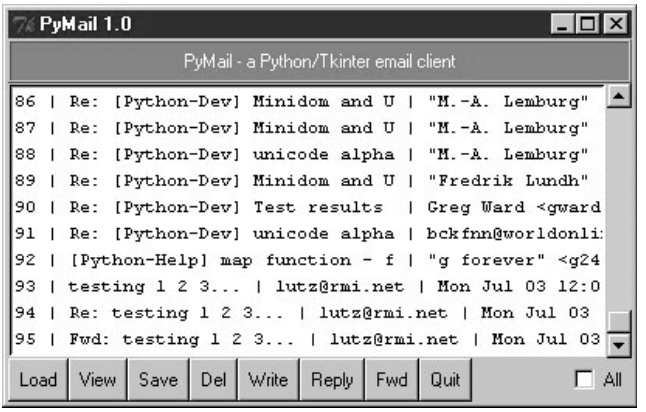


Рис. 11.29. Список почты PyMailGui после отправки и загрузки

Имейте в виду, что PyMailGui выполняется на локальном компьютере, но сообщения, которые вы видите в списке главного окна, фактически находятся в почтовом ящике на машине сервера. Каждый раз, когда мы нажимаем Load, PyMailGui загружает с сервера на ваш компьютер вновь поступившую почту, но не удаляет ее с сервера. Три сообщения, которые мы только что написали (93–95), появятся и в любой другой почтовой программе, которой вы воспользуетесь со своей учетной записью (например, в Outlook). PyMailGui не удаляет сообщения при загрузке, а просто записывает их в память вашего компьютера для обработки. Если теперь выделить сообщение 95 и нажать View, можно увидеть отправленное переадресованное сообщение, как на рис. 11.30. В действительности это сообщение ушло с моей машины на удаленный почтовый сервер и было загружено с него в список Python, из которого и показано.

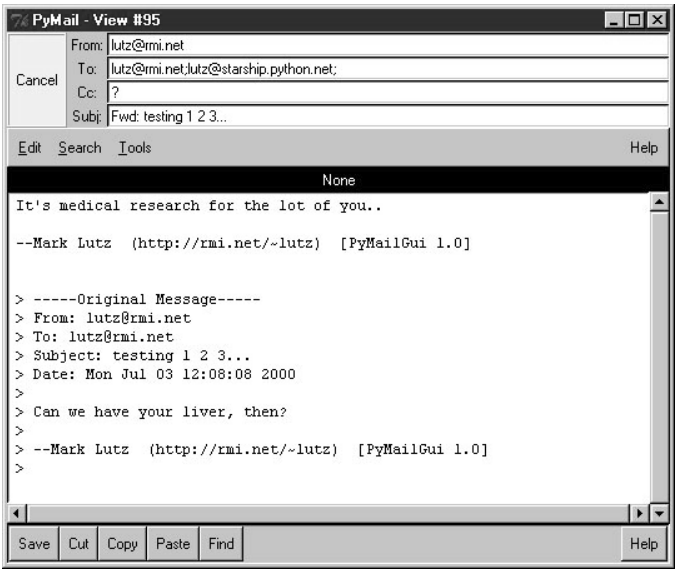


Рис. 11.30. Просмотр переадресованной почты в PyMailGui

Рис. 11.31 показывает необработанный текст переадресованного сообщения; чтобы вывести эту форму, нужно сделать двойной щелчок по записи в главном окне. Форматированное отображение на рис. 11.30 просто извлекает элементы текста, показываемого в форме, которая отображает необработанный текст.

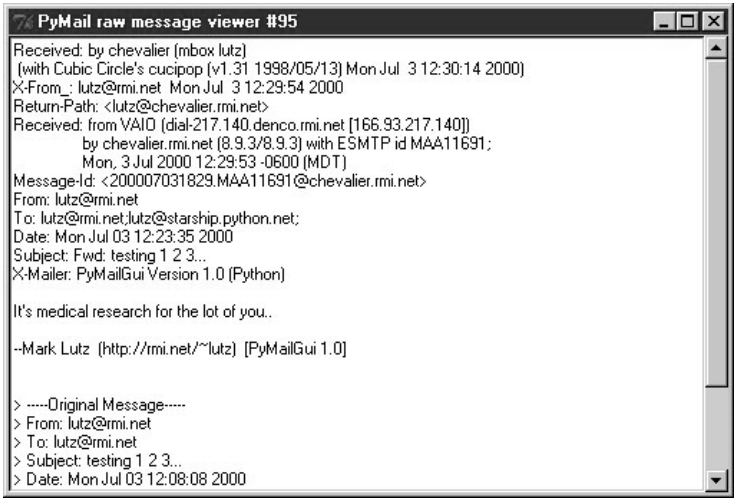


Рис. 11.31. Просмотр в PyMailGui исходного текста переадресованного сообщения

Об обратных адресах

Тонкое замечание: технически адрес «To:» в ответе на сообщение образуется из данных, возвращаемых вызовом стандартной библиотеки вида `hdrs.getaddr('From')` – через интерфейс модуля `rfc822`, который автоматически разбирает и форматирует адрес отправителя исходного сообщения, – плюс кавычки, добавляемые в некоторых редких случаях.

Посмотрите на функцию `onReplyMail` в листингах кода. Этот библиотечный вызов возвращает пару (*полное имя*, *адрес электронной почты*), выделенные из строки заголовка сообщения «From:». Например, если в сообщении первый заголовок «From:» содержит строку

```
'joe@spam.net (Joe Blow)'
```

то вызов `hdrs.getaddr('From')` даст пару `('Joe Blow', 'joe@spam.net')` с пустой строкой имени, если оно не указано в адресной строке исходного отправителя. Если же заголовок содержит

```
'Joe Blow <joe@spam.net>'
```

то вызов возвратит тот же самый результирующий набор.

К несчастью, в версии Python 1.5.2 модуль `rfc822` содержал ошибку, из-за которой этот вызов действовал не всегда корректно: функция `getaddr` дает ложные результаты, если в части имени, содержащей полное имя, есть запятая (например, `"Blow, Joe"`). Возможно, в Python 2.0 эта ошибка исправлена, но чтобы справиться с ней в более ранних версиях, PyMailGui помещает имя в явные кавычки, прежде чем вставить его в конечный адрес *полное-имя*<*адрес-email*>, используемый в строке ответа «To:». Например, вот четыре типичных адреса «From:» и ответный адрес «To:», который PyMailGui генерирует для каждого из них (после =>):

```
joe@spam.net  => <joe@spam.net>
Joe Blow <joe@spam.net>    => Joe Blow <joe@spam.net>
joe@spam.net (Joe Blow)    => Joe Blow <joe@spam.net>
"Blow, Joe" <joe@spam.net> => "Blow, Joe" <joe@spam.net>
```

Без дополнительных кавычек вокруг имени в последнем из примеров запятая ввела бы в заблуждение мой сервер SMTP, который счел бы, что у письма два получателя – *Blow@rmi.net* и *Joe <joe@spam.net>* (первый ошибочно получает доменное имя моего провайдера, поскольку предполагается, что это локальный пользователь). Лишние кавычки не повредят, если в последующих версиях ошибка будет устранена.

Менее сложным альтернативным решением (котрое позднее будет использовано в программе с именем PyMailCgi) является использование исходного адреса «From:» в точности как «To:» в ответе. Библиотечный вызов вида `hdrs.get('From')` возвращает адрес отправителя дословно со всеми кавычками, не пытаясь выделить его составляющие.

В существующем виде схема PyMailGui составления обратного адреса действовала для всех сообщений, на которые мне когда-либо приходилось отвечать, но возможно, что ее понадобится скорректировать для каких-то особых форматов адресов или новых версий Python. Я много проверял и пользовался этой про-

граммой, но мало ли что может произойти в Сети, несмотря на наличие стандартов почтовых адресов. Официально заявляю, что любые ошибки, которые вы обнаружите в программе, в действительности являются замаскированными упражнениями для читателей (во всяком случае я не говорю, что «баги» – это «фичи»).

Сохранение и удаление электронной почты

К настоящему времени мы осветили все, кроме двух кнопок обработки в главном окне и флажка All. PyMailGui позволяет сохранять почтовые сообщения в локальных текстовых файлах и удалять навсегда сообщения с сервера, чтобы не видеть их при следующем обращении к своей учетной записи. Более того, можно сохранить или удалить одно сообщение или все сообщения, выводимые в списке главного окна:

- Чтобы сохранить одно сообщение, выделите его в списке главного окна и нажмите кнопку Save.
- Чтобы сохранить все сообщения в списке за один шаг, щелкните по флажку All в правом нижнем углу главного окна, а затем нажмите Save.

Операции удаления запускаются таким же способом, только нажимать нужно кнопку Del. При обычной работе я в конце удаляю почту, которая мне не интересна, и сохраняю и удаляю важные сообщения. Операции сохранения записывают исходный текст одного или более сообщений в локальный текстовый файл, выбираемый в окне диалога, показанном на рис. 11.32.

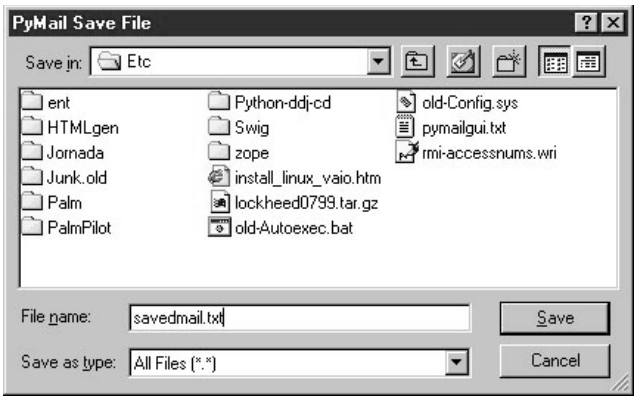


Рис. 11.32. Диалог PyMailGui сохранения почты

Технически при сохранении сообщения его исходный текст всегда дописывается к выбранному файлу: файл открывается в режиме `'a'`, при этом файл при необходимости создается, и запись осуществляется в его конец. Операция сохранения также запоминает каталог, который открывался последним: с него начинает навигацию диалог выбора файла при очередном нажатии Save.

Операции удаления тоже могут применяться к одному или ко всем сообщениям. Однако в отличие от других операций, запросы на удаление просто помещаются в очередь для исполнения в дальнейшем. Фактическое удаление сообщений с почтового сервера происходит только при выходе из PyMailGui. Например, если отобрать некоторые сообщения для удаления и нажать в главном окне кнопку Quit, появляется стандартный диалог подтверждения (рис. 11.33).

Если подтвердить запрос на выход, появляется второй диалог (рис. 11.34) с просьбой подтвердить удаление сообщений в очереди. Если нажать здесь No, удаление не производится и PyMailGui просто завершает работу. Если выбрать Yes, PyMailGui в последний раз порождает новый поток для отправки почтовому серверу запросов на удаление для всех сообщений, отобранных для этого во время сеанса работы. Во время выполнения потока удаления показывается окно состояния ожидания. По окончании этого потока PyMailGui также завершается.

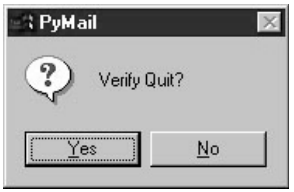


Рис. 11.33. Запрос на подтверждение при выходе из PyMailGui

Рис. 11.34. Запрос подтверждения удаления при выходе из PyMailGui

По умолчанию почта никогда не удаляется: те же самые сообщения будут показаны при очередном запуске PyMailGui. Почта удаляется с сервера, только если потребовать этого, сообщения удаляются только при выходе из программы и только при подтверждении в последнем показанном всплывающем окне (это последняя возможность избежать удаления почты навсегда).

Номера сообщений в POP

Такой способ удаления почты может показаться окольным, но он приспособлен к особенности интерфейса POP. В POP каждому сообщению присваивается последовательный номер начиная с 1. Эти номера и передаются серверу для получения и удаления сообщений. Нет ничего плохого, если в то время, когда выводится результат предыдущей загрузки, поступают новые письма – им будут присвоены более высокие номера по сравнению с теми, которые выведены у клиента. Но если удалить сообщение, находящееся в середине почтового ящика, номера всех сообщений, расположенных после удаленного, изменятся (они уменьшатся на единицу). Это означает, что некоторые номера сообщений могут стать недействительными, если удаление осуществляется во время просмотра ранее загруженной почты (попытка удалить письмо с номером N могла бы в действительности привести к удалению сообщения с номером N+1!).

Для решения этой проблемы можно было бы корректировать в PyMailGui все отображаемые номера. Однако проще отложить удаление на будущее. Заметьте, что если одновременно выполнять несколько экземпляров PyMailGui, нельзя удалять сообщения в одном экземпляре, а потом в другом, потому что может возникнуть путаница с номерами. Плохие результаты могут получиться и при запуске, скажем, Outlook одновременно с сеансом PyMailGui, но конечный результат такой комбинации зависит от того, как обрабатываются удаления вторым почтовым клиентом. В принципе, можно добавить в PyMailGui запрет одновременного запуска нескольких его экземпляров, но это оставляется в качестве упражнения.

Окна и сообщения о статусе

Прежде чем завершить этот раздел, хочу подчеркнуть, что PyMailGui в действительности разработан как *многооконный* интерфейс, что может быть неочевидно из приводившихся снимков экранов. Например, на рис. 11.35 показан PyMailGui с главным окном списка, подсказкой и тремя окнами просмотра писем. Все эти окна не модаль-

ны, то есть действуют независимо и не мешают выбору других окон. Под Linux этот интерфейс выглядит несколько иным образом, но действует так же.

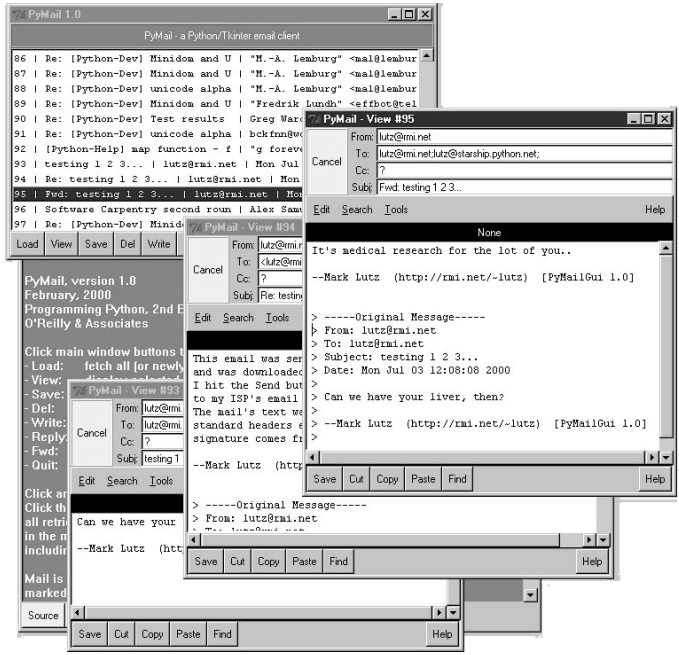


Рис. 11.35. Несколько окон и текстовых редакторов PyMailGui

В целом одновременно может быть открыто любое количество окон для просмотра или составления сообщений, между которыми можно осуществлять операции «удалить и вставить». Это важно, поскольку PyMailGui должен обеспечить наличие в каждом окне *отдельного* объекта текстового редактора. Если бы объект текстового редактора был глобальным или использовал глобальные переменные, могло оказаться, что в каждом окне отображается один и тот же текст (а операции отправки могли бы привести к отправке текста из другого окна). Чтобы избежать этого, PyMailGui создает и вставляет новые экземпляры `TextEditor` для каждого нового окна просмотра или составления сообщения и связывает новый редактор с обработчиком обратного вызова кнопки Send, обеспечивая получение нужного текста.

Наконец, во время своей работы PyMailGui выводит целый ряд сообщений о статусе, но увидеть их можно, только если запустить программу из командной строки системы (например, окна DOS в Windows или xterm в Linux) или двойным щелчком по значку с именем файла (главным сценарием является *.pyw*, а не *.pyw*). В Windows эти сообщения не будут видны при запуске из другой программы, например из GUI-панелей запуска PyDemos или PyGadgets. Эти сообщения о статусе содержат информацию о сервере, показывают состояние загрузки почты и трассируют порождаемые попутно потоки загрузки, сохранения и удаления. Если вы хотите увидеть сообщения PyMailGui, запустите его из командной строки и наблюдайте:

```
C:\...\PP2E\Internet\Email>python PyMailGui.py
load start
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <594100005a655e39@chevalier>
```

```
('+OK 5 messages (8470 octets)', ['1 709', '2 1136', '3 998', '4 2679',
'5 2948'], 38)
There are 5 mail messages in 8470 bytes
Retrieving: 1 2 3 4 5
load exit
thread exit caught
send start
Connecting to mail... ['<lutz@rmi.net>']
send exit
thread exit caught
```

Можно также сделать двойной щелчок по файлу *PyMailGui.py* в GUI менеджера файлов и наблюдать появившееся окно консоли DOS в Windows; это окно показано на рис. 11.36.

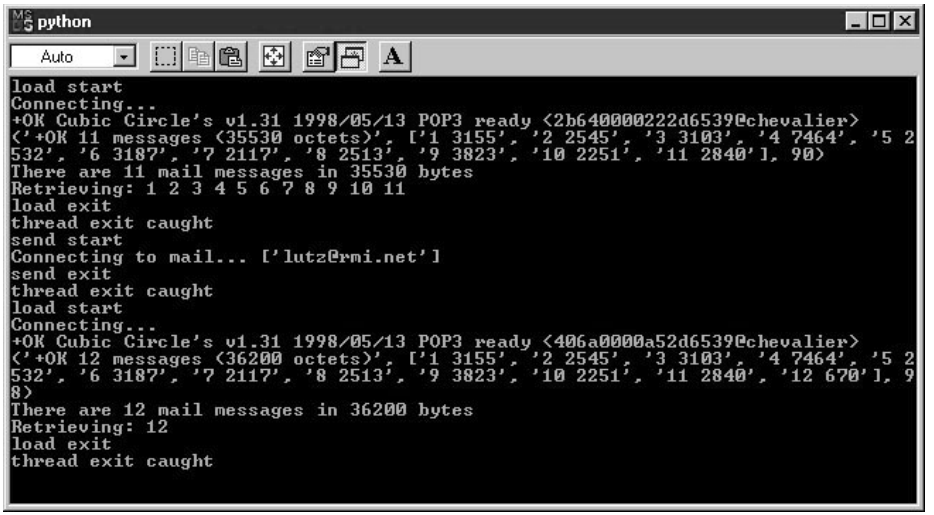


Рис. 11.36. Окно консоли с сообщениями о статусе PyMailGui

Сообщения о статусе PyMailGui показывают, какое сообщение загружается в данный момент (к строке «Retrieving:» при загрузке каждого сообщения добавляется номер нового письма), и потому более информативны, чем всплывающее окно ожидания.

Реализация PyMailGui

И наконец мы добрались до кода. В действительности здесь только два новых модуля: один из них содержит текст подсказки, а другой реализует систему.

На практике PyMailGui весьма выигрывает от повторного использования модулей, которые мы написали раньше и не будем здесь повторять: `pymail` для операций загрузки и удаления почты, `mailconfig` для параметров почты, `TextEditor` из раздела, посвященного GUI, для вывода и редактирования текста сообщений и т. д. Кроме того, здесь используются такие стандартные модули Python, как `poplib`, `smtplib` и `rfc822`, скрывающие большую часть действий по пересылке байтов по Сети и извлечению частей сообщений. `Tkinter` реализует компоненты GUI с поддержкой переносимости.

Модуль с текстом подсказки

В результате повторного использования кода PyMailGui реализует достаточно богатую функциями почтовую программу примерно в 500 строках кода плюс один вспомогательный модуль. Этот вспомогательный модуль приведен в примере 11.22 и используется только для определения строки текста подсказки, чтобы избежать загромождения файла с основным сценарием.

Пример 11.22. PP2E\Internet\Email\PyMailGuiHelp.py

```
##############################################################################
# Строка текста подсказки PyMailGui размещена в отдельном модуле, только
# чтобы не отвлекать внимание от исполняемого кода. В данном случае этот текст
# помещается в простое окно с прокруткой; в будущем можно было бы использовать
# файл HTML и открывать его в веб-броузере (например, выполнять "netscape
# help.html" или команду DOS "start help.html" через вызов os.system
##############################################################################

# Текст должен быть узким для вывода в информационных окнах Linux;
# сейчас используется scrolledtext с кнопками;

helptext = """
PyMail, version 1.0
February, 2000
Programming Python, 2nd Edition
O'Reilly & Associates

Click main window buttons to process email:
- Load:\t fetch all (or newly arrived) POP mail from server
- View:\t display selected message nicely formatted
- Save:\t write selected (or all) emails to a chosen file
- Del:\t mark selected (or all) email to be deleted on exit
- Write:\t compose a new email message, send it by SMTP
- Reply:\t compose a reply to selected email, send it by SMTP
- Fwd:\t compose a forward of selected email, send by SMTP
- Quit:\t exit PyMail, delete any marked emails from server

Click an email in the main window's listbox to select it.
Click the "All" checkbox to apply Save or Del buttons to
all retrieved emails in the list.  Double-click on an email
in the main window's listbox to view the mail's raw text,
including mail headers not shown by the View button.

Mail is removed from POP servers on exit only, and only mails
marked for deletion with the Del button are removed, if and
only if you verify the deletion in a confirmation popup.

Change the mailconfig.py module file on your own machine to
reflect your email server names, user name, email address,
and optional mail signature line added to all composed mails.
Miscellaneous hints:

- Passwords are requested if needed, and not stored by PyMail.
- You may put your password in a file named in mailconfig.py.
- Use ';' between multiple addresses in "To" and "Cc" headers.
- Reply and Fwd automatically quote the original mail text.
- Save pops up a dialog for selecting a file to hold saved mails.
- Load only fetches newly-arrived email after the first load.

This client-side program currently requires Python and Tkinter.
It uses Python threads, if installed, to avoid blocking the GUI.
```

```
Sending and loading email requires an Internet connection.
"""

if __name__ == '__main__':
    print helptext                      # в stdout при автономном выполнении
    raw_input('Press Enter key')        # пауза в окне консоли DOS
```

Основной модуль

И вот, наконец, главный сценарий PyMailGui – файл, который запускает систему (см. пример 11.23). Я уже рассказал, что он делает и почему, поэтому изучение листинга этого кода и комментариев в нем для более глубокого рассмотрения оставляется читателю в качестве упражнения. Python настолько близок к псевдокоду, что дополнительный рассказ будет, вероятно, излишним.

Хотя я регулярно пользуюсь этим примером в повседневной работе в том виде, как он здесь представлен, его можно расширить. Например:

- Удаленные сообщения можно пометить графически.
- Почтовые вложения можно автоматически отображать, выделять, раскодировать и открывать щелчком с помощью рассмотренных выше средств Python для обработки сообщений из нескольких частей.
- Состояние загрузки можно показывать более информативным образом во время операций получения почты, если обновлять индикатор после каждого полученного письма (например, периодически изменяя размер прямоугольника, рисуемого на холсте во всплывающем окне).
- URL гиперссылок в сообщениях можно выделять зрительно и при щелчке по ним автоматически запускать веб-броузер, используя средства запуска программ, рассмотренные в частях книги, посвященных GUI и системным инструментам.
- Поскольку сообщения телеконференций Интернета аналогичны по структуре сообщениям электронной почты (строки заголовков плюс текст тела: смотрите пример `nntplib` в следующем разделе), этот сценарий в принципе можно расширить так, чтобы он выводил как почтовые сообщения, так и статьи телеконференций. Классифицировать такое видоизменение, как толковое обобщение или зверский взлом кода, предоставляется в качестве отдельного упражнения.
- В данном сценарии все же используется нестандартный, но обычно безвредный формат даты отправки, о чем говорилось выше во врезке; можно тривиальным образом импортировать согласующуюся со стандартом функцию форматирования даты из модуля `pymail`.
- Во время передачи почты PyMailGui выводит диалоговое окно ожидания, которое фактически отключает остальную часть GUI. Это сделано умышленно, чтобы не заниматься проблемами синхронизации. В принципе, однако, система могла бы позволить потокам почтовых операций перекрываться по времени (например, разрешить пользователю отправлять новые сообщения во время загрузки почты). Поскольку каждая из пересылок выполняется на собственном сокете, PyMailGui не нужно блокировать другие операции во время пересылки. Это можно было бы реализовать с помощью периодических событий Tkinter `after`, которые проверяли бы состояние текущей пересылки. Пример перекрывающихся потоков пересылки данных смотрите в сценариях PyFtpGui ранее в этой главе.

И так далее. Поскольку это программное обеспечение с открытым кодом, оно также открыто для дополнения. Возможные упражнения в этой области определяются вашим воображением.

Пример 11.23. PP2E\Internet\Email\PyMailGui.py

```
############################################################################################
# PyMailGui 1.0 - почтовый клиент Python/Tkinter. Добавляет GUI на основе Tkinter к функциям
# сценария pymail. Работает с серверами POP/SMTP, используя сокеты машины, на которой
# выполняется этот сценарий. Используются потоки, если загрузка, отправка и удаление
# должны выполняться без блокировки; в Windows потоки являются стандартом. Обновление GUI
# происходит только в главном потоке (Windows). Повторно использует и встраивает объект
# класса TextEditor. Для просмотра сообщений о статусе запускать из командной строки.
# См. замечания по использованию в тексте подсказки в PyMailGuiHelp.py.
# Добавить: поддержку вложений, оттенять удаления.
############################################################################################

# получить сервисы
import pymail, mailconfig
import rfc822, StringIO, string, sys
from Tkinter       import *
from tkFileDialog  import asksaveasfilename, SaveAs
from tkMessageBox  import showinfo, showerror, askyesno
from PP2E.Gui.TextEditor.textEditor import TextEditorComponentMinimal

# выполнение без потоков
try:                                    # возбудить ImportError для
    import thread                       # выполнения с блокировкой gui
except ImportError:                     # не появляется окон ожидания
    class fakeThread:
        def start_new_thread(self, func, args):
            apply(func, args)
    thread = fakeThread()

# init global/module vars
msgList       = []                      # список полученных текстов e-mail
toDelete      = []                      # номера сообщений, удаляемых при выходе
listBox       = None                    # прокручиваемый список сообщений главного окна
rootWin       = None                    # главное окно этой программы
allModeVar    = None                    # значение флажка режима All
threadExitVar = 0                       # для сигнала о завершении дочернего потока
debugme       = 0                       # включить дополнительные сообщения о статусе

mailserver = mailconfig.popservername   # откуда читать почту pop
mailuser   = mailconfig.popusername     # сервер smtp тоже в mailconfig
mailpswd   = None                       # пароль pop из файла или через диалог, как здесь
#mailfile  = mailconfig.savemailfile    # здесь через диалог выбора файла

def fillIndex(msgList):
    # заполнить целиком главное окно списка
    listBox.delete(0, END)
    count = 1
    for msg in msgList:
        hdrs = rfc822.Message(StringIO.StringIO(msg))
        msginfo = '%02d' % count
        for key in ('Subject', 'From', 'Date'):
            if hdrs.has_key(key): msginfo = msginfo + ' | ' + hdrs[key][:30]
        listBox.insert(END, msginfo)
        count = count+1
    listBox.see(END)                    # показать самую свежую почту=последняя строка

def selectedMsg():
    # получить сообщение, выбранное в главном окне списка
    # print listBox.curselection()
```

```
    if listBox.curselection() == ():
        return 0                                    # пустой кортеж:ничего не выбрано
    else:                                           # или индекс начиная с нуля
        return eval(listBox.curselection()[0]) + 1  # в кортеже из одной строки

def waitForThreadExit(win):
    import time
    global threadExitVar            # в главном потоке следить за глобальной переменной
    delay = 0.0                     # 0.0=в Win98 не нужно засыпать (но жрет cpu)
    while not threadExitVar:
        win.update()                # отправлять новые события GUI во время ожидания
        time.sleep(delay)           # если нужно, заснуть, чтобы выполнялись другие потоки
    threadExitVar = 0               # одновременно активно не более одного дочернего потока

def busyInfoBoxWait(message):
    # вывести окно ожидания, ждать выхода из потока
    # поток событий главного gui продолжает жить во время ожидания
    # в таком виде возврат только после завершения потока
    # popup.wait_variable(threadExitVar) тоже может работать

    popup = Toplevel()
    popup.title('PyMail Wait')
    popup.protocol('WM_DELETE_WINDOW', lambda:0)       # игнорировать закрытие
    label = Label(popup, text=message+'...')
    label.config(height=10, width=40, cursor='watch')  # курсор занятости
    label.pack()
    popup.focus_set()                                  # захват приложения
    popup.grab_set()                                   # ждать выхода из потка
    waitForThreadExit(popup)                           # gui жив во время ожидания
    print 'thread exit caught'
    popup.destroy()

def loadMailThread():
    # загружать почту при обработке главным потоком событий gui
    global msgList, errInfo, threadExitVar
    print 'load start'
    errInfo = ''
    try:
        nextnum = len(msgList) + 1
        newmail = pymail.loadmessages(mailserver, mailuser, mailpswd, nextnum)
        msgList = msgList + newmail
    except:
        exc_type, exc_value = sys.exc_info()[:2]       # исключительная ситуация потока
        errInfo = '\n' + str(exc_type) + '\n' + str(exc_value)
    print 'load exit'
    threadExitVar = 1   # сигнал главному потоку

def onLoadMail():
    # загрузить всю (или новую) почту pop
    getpassword()
    thread.start_new_thread(loadMailThread, ())
    busyInfoBoxWait('Retrieving mail')
    if errInfo:
        global mailpswd                                # очистить пароль для повторного ввода
        mailpswd = None
        showerror('PyMail', 'Error loading mail\n' + errInfo)
    fillIndex(msgList)

def onViewRawMail():
    # просмотр выбранного сообщения – исходный текст со строками заголовка
```

```
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        text = msgList[msgnum-1]                    # поместить в ScrolledText
        from ScrolledText import ScrolledText
        window  = Toplevel()
        window.title('PyMail raw message viewer       #' + str(msgnum))
        browser = ScrolledText(window)
        browser.insert('0.0', text)
        browser.pack(expand=YES, fill=BOTH)

def onViewFormatMail():
    # просмотр выбранного сообщения - показ в форматированном виде
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        mailtext = msgList[msgnum-1]                # поместить в форму TextEditor
        textfile = StringIO.StringIO(mailtext)
        headers  = rfc822.Message(textfile)         # удаляет строки заголовков
        bodytext = textfile.read()                  # остальное - тело сообщения
        editmail('View #%d' % msgnum,
                 headers.get('From', '?'),
                 headers.get('To', '?'),
                 headers.get('Subject', '?'),
                 bodytext,
                 headers.get('Cc', '?'))

# использовать объекты, сохраняющие прежний каталог для следующего
# выбора вместо простого диалога asksaveasfilename()

saveOneDialog = saveAllDialog = None

def myasksaveasfilename_one():
    global saveOneDialog
    if not saveOneDialog:
        saveOneDialog = SaveAs(title='PyMail Save File')
    return saveOneDialog.show()

def myasksaveasfilename_all():
    global saveAllDialog
    if not saveAllDialog:
        saveAllDialog = SaveAs(title='PyMail Save All File')
    return saveAllDialog.show()

def onSaveMail():
    # сохранить выбранное сообщение в файле
    if allModeVar.get():
        mailfile = myasksaveasfilename_all()
        if mailfile:
            try:
                # возможно, это должен быть поток
                for i in range(1, len(msgList)+1):
                    pymail.savemessage(i, mailfile, msgList)
            except:
                showerror('PyMail', 'Error during save')
    else:
        msgnum = selectedMsg()
        if not (1 <= msgnum <= len(msgList)):
```

```
            showerror('PyMail', 'No message selected')
        else:
            mailfile = myasksaveasfilename_one()
            if mailfile:
                try:
                    pymail.savemessage(msgnum, mailfile, msgList)
                except:
                    showerror('PyMail', 'Error during save')

def onDeleteMail():
    # пометить выбранное сообщение для удаления при выходе
    global toDelete
    if allModeVar.get():
        toDelete = range(1, len(msgList)+1)
    else:
        msgnum = selectedMsg()
        if not (1 <= msgnum <= len(msgList)):
            showerror('PyMail', 'No message selected')
        elif msgnum not in toDelete:
            toDelete.append(msgnum)                          # неудача при повторении в списке

def sendMailThread(From, To, Cc, Subj, text):
    # отправка почты одновременно с обработкой событий gui главным потоком
    global errInfo, threadExitVar
    import smtplib, time
    from mailconfig import smtpservername
    print 'send start'

    date  = time.ctime(time.time())
    Cchdr = (Cc and 'Cc: %s\n' % Cc) or ''
    hdrs  = ('From: %s\nTo: %s\n%sDate: %s\nSubject: %s\n' % (From, To, Cchdr, date, Subj))
    hdrs  = hdrs + 'X-Mailer: PyMailGui Version 1.0 (Python)\n\n'

    Ccs = (Cc and string.split(Cc, ';')) or []           # некоторые серверы отклоняют ['']
    Tos = string.split(To, ';') + Ccs                    # cc: строка заголовка и список To
    Tos = map(string.strip, Tos)                         # в некоторых адресах могут быть ','
    print 'Connecting to mail...', Tos                   # удалить пробелы по концам адресов

    errInfo = ''
    failed  = {}                                         # smtplib может вызывать исключения
    try:                                                 # или возвращать словарь неудачных To
        server = smtplib.SMTP(smtpservername)
        failed = server.sendmail(From, Tos, hdrs + text)
        server.quit()
    except:
        exc_type, exc_value = sys.exc_info()[:2]   # исключение потока
        excinfo = '\n' + str(exc_type) + '\n' + str(exc_value)
        errInfo = 'Error sending mail\n' + excinfo
    else:
        if failed: errInfo = 'Failed recipients:\n' + str(failed)

    print 'send exit'
    threadExitVar = 1                                    # сигнал главному потоку

def sendMail(From, To, Cc, Subj, text):
    # отправить законченные сообщения
    thread.start_new_thread(sendMailThread, (From, To, Cc, Subj, text))
    busyInfoBoxWait('Sending mail')
    if errInfo:
        showerror('PyMail', errInfo)
```

```
def onWriteReplyFwdSend(window, editor, hdrs):
    # нажатие кнопки отправки окна редактирования почты
    From, To, Cc, Subj = hdrs
    sendtext = editor.getAllText()
    sendMail(From.get(), To.get(), Cc.get(), Subj.get(), sendtext)
    if not errInfo:
        window.destroy()    # else keep to retry or save

def editmail(mode, From, To='', Subj='', origtext='', Cc=''):
    # создать новое окно редактирования/просмотра почты
    win = Toplevel()
    win.title('PyMail - '+ mode)
    win.iconname('PyMail')
    viewOnly = (mode[:4] == 'View')

    # поля для ввода заголовков
    frm =  Frame(win); frm.pack( side=TOP,   fill=X)
    lfrm = Frame(frm); lfrm.pack(side=LEFT,  expand=NO,  fill=BOTH)
    mfrm = Frame(frm); mfrm.pack(side=LEFT,  expand=NO,  fill=NONE)
    rfrm = Frame(frm); rfrm.pack(side=RIGHT, expand=YES, fill=BOTH)
    hdrs = []
    for (label, start) in [('From:', From),
                           ('To:',   To),                     # при отправке важен порядок
                           ('Cc:',   Cc),
                           ('Subj:', Subj)]:
        lab = Label(mfrm, text=label, justify=LEFT)
        ent = Entry(rfrm)
        lab.pack(side=TOP, expand=YES, fill=X)
        ent.pack(side=TOP, expand=YES, fill=X)
        ent.insert('0', start)
        hdrs.append(ent)

    # кнопки send, cancel (нужен новый редактор)
    editor = TextEditorComponentMinimal(win)
    sendit = (lambda w=win, e=editor, h=hdrs: onWriteReplyFwdSend(w, e, h))

    for (label, callback) in [('Cancel', win.destroy), ('Send', sendit)]:
        if not (viewOnly and label == 'Send'):
            b = Button(lfrm, text=label, command=callback)
            b.config(bg='beige', relief=RIDGE, bd=2)
            b.pack(side=TOP, expand=YES, fill=BOTH)

    # редактор текста тела: pack последним=обрезается первым
    editor.pack(side=BOTTOM)                            # могут быть многие редакторы
    if (not viewOnly) and mailconfig.mysignature:       # добавить текст автоматической подписи,
        origtext = ('\n%s\n' % mailconfig.mysignature) + origtext
    editor.setAllText(origtext)

def onWriteMail():
    # составить новое сообщение
    editmail('Write', From=mailconfig.myaddress)

def quoteorigtext(msgnum):
    origtext = msgList[msgnum-1]
    textfile = StringIO.StringIO(origtext)
    headers  = rfc822.Message(textfile)                 # удаляет строки заголовков
    bodytext = textfile.read()                          # остальное - тело сообщения
    quoted   = '\n-----Original Message-----\n'
    for hdr in ('From', 'To', 'Subject', 'Date'):
        quoted = quoted + ( '%s: %s\n' % (hdr, headers.get(hdr, '?')) )
```

```
    quoted   = quoted + '\n' + bodytext
    quoted   = '\n' + string.replace(quoted, '\n', '\n> ')
    return quoted

def onReplyMail():
    # ответ на выбранное сообщение
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        text = quoteorigtext(msgnum)
        hdrs = rfc822.Message(StringIO.StringIO(msgList[msgnum-1]))
        toname, toaddr = hdrs.getaddr('From')
        if toname and ',' in toname: toname = '"%s"' % toname
        To   = '%s <%s>' % (toname, toaddr)
        From = mailconfig.myaddress or ('%s <%s>' % hdrs.getaddr('To'))
        Subj = 'Re: ' + hdrs.get('Subject', '(no subject)')
        editmail('Reply', From, To, Subj, text)

def onFwdMail():
    # переадресация выбранного сообщения
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        text = quoteorigtext(msgnum)
        hdrs = rfc822.Message(StringIO.StringIO(msgList[msgnum-1]))
        From = mailconfig.myaddress or ('%s <%s>' % hdrs.getaddr('To'))
        Subj = 'Fwd: ' + hdrs.get('Subject', '(no subject)')
        editmail('Forward', From, '', Subj, text)

def deleteMailThread(toDelete):
    # удаление почты одновременно с обработкой событий gui главным потоком
    global errInfo, threadExitVar
    print 'delete start'
    try:
        pymail.deletemessages(mailserver, mailuser, mailpswd, toDelete, 0)
    except:
        exc_type, exc_value = sys.exc_info()[:2]
        errInfo = '\n' + str(exc_type) + '\n' + str(exc_value)
    else:
        errInfo = ''
    print 'delete exit'
    threadExitVar = 1   # signal main thread

def onQuitMail():
    # выход из почтовой программы, теперь удалять
    if askyesno('PyMail', 'Verify Quit?'):
        if toDelete and askyesno('PyMail', 'Really Delete Mail?'):
            getpassword()
            thread.start_new_thread(deleteMailThread, (toDelete,))
            busyInfoBoxWait('Deleting mail')
            if errInfo:
                showerror('PyMail', 'Error while deleting:\n' + errInfo)
            else:
                showinfo('PyMail', 'Mail deleted from server')
        rootWin.quit()

def askpassword(prompt, app='PyMail'):    # getpass.getpass использует stdin, а не GUI
    win = Toplevel()                      # tkSimpleDialog.askstring повторяет ввод
```

```
    win.title(app + ' Prompt')
    Label(win, text=prompt).pack(side=LEFT)
    entvar = StringVar()
    ent = Entry(win, textvariable=entvar, show='*')
    ent.pack(side=RIGHT, expand=YES, fill=X)
    ent.bind('<Return>', lambda event, savewin=win: savewin.destroy())
    ent.focus_set(); win.grab_set(); win.wait_window()
    win.update()
    return entvar.get()                  # элемента ent теперь нет

def getpassword():
    # если глобальный пароль pop неизвестен,
    # установить из файла клиента или из диалога
    global mailpswd
    if mailpswd:
        return
    else:
        try:
            localfile = open(mailconfig.poppasswdfile)
            mailpswd  = localfile.readline()[:-1]
            if debugme: print 'local file password', repr(mailpswd)
        except:
            prompt    = 'Password for %s on %s?' % (mailuser, mailserver)
            mailpswd  = askpassword(prompt)
            if debugme: print 'user input password', repr(mailpswd)

def decorate(rootWin):
    # настройка главного окна менеджером окон
    rootWin.title('PyMail 1.0')
    rootWin.iconname('PyMail')
    rootWin.protocol('WM_DELETE_WINDOW', onQuitMail)

def makemainwindow(parent=None):
    # создать главное окно
    global rootWin, listBox, allModeVar
    if parent:
        rootWin = Frame(parent)                # прикрепить к родителю
        rootWin.pack(expand=YES, fill=BOTH)
    else:
        rootWin = Tk()                         # предполагая автономное выполнение
        decorate(rootWin)

    # добавить внизу основные кнопки
    frame1 = Frame(rootWin)
    frame1.pack(side=BOTTOM, fill=X)
    allModeVar = IntVar()
    Checkbutton(frame1, text="All", variable=allModeVar).pack(side=RIGHT)
    actions = [ ('Load',  onLoadMail),  ('View',  onViewFormatMail),
                ('Save',  onSaveMail),  ('Del',   onDeleteMail),
                ('Write', onWriteMail), ('Reply', onReplyMail),
                ('Fwd',   onFwdMail),   ('Quit',  onQuitMail) ]
    for (title, callback) in actions:
        Button(frame1, text=title, command=callback).pack(side=LEFT, fill=X)

    # добавить окно списка и полосу прокрутки
    frame2  = Frame(rootWin)
    vscroll = Scrollbar(frame2)
    fontsz  = (sys.platform[:3] == 'win' and 8) or 10
    listBox = Listbox(frame2, bg='white', font=('courier', fontsz))
```

```
    # связать окно списка и полосу прокрутки
    vscroll.config(command=listBox.yview, relief=SUNKEN)
    listBox.config(yscrollcommand=vscroll.set, relief=SUNKEN, selectmode=SINGLE)
    listBox.bind('<Double-1>', lambda event: onViewRawMail())
    frame2.pack(side=TOP, expand=YES, fill=BOTH)
    vscroll.pack(side=RIGHT, fill=BOTH)
    listBox.pack(side=LEFT, expand=YES, fill=BOTH)
    return rootWin

# загрузить строку блока текста
from PyMailGuiHelp import helptext

def showhelp(helptext=helptext, appname='PyMail'):   # показать текст подсказки
    from ScrolledText import ScrolledText            # в немодальном диалоге
    new  = Toplevel()                                # создать новое всплывающее окно
    bar  = Frame(new)                                # пакуется первым=обрезается последним
    bar.pack(side=BOTTOM, fill=X)
    code = Button(bar, bg='beige', text="Source", command=showsource)
    quit = Button(bar, bg='beige', text="Cancel", command=new.destroy)
    code.pack(pady=1, side=LEFT)
    quit.pack(pady=1, side=LEFT)
    text = ScrolledText(new)                         # добавить Text + полосу прокрутки
    text.config(font='system',  width=70)            # слишком большой для showinfo
    text.config(bg='steelblue', fg='white')          # удалить по кнопке или return
    text.insert('0.0', helptext)
    text.pack(expand=YES, fill=BOTH)
    new.title(appname + " Help")
    new.bind("<Return>", (lambda event, new=new: new.destroy()))

def showsource():
    # замысловато, но открыто
    try:                                             # как getfile.cgi в веб
        source = open('PyMailGui.py').read()         # в текущем каталоге или ниже?
    except:
        try:                                         # или использовать find.find(f)[0],
            import os                                # $PP2EHOME, guessLocation
            from PP2E.Launcher import findFirst      # или породить pyedit с аргументом
            here   = os.curdir
            source = open(findFirst(here, 'PyMailGui.py')).read()
        except:
            source = 'Sorry - cannot find my source file'
    subject = 'Main script [see also: PyMailGuiHelp, pymail, mailconfig]'
    editmail('View Source Code', 'PyMailGui', 'User', subject, source)

def container():
    # добавить кнопку help путем прикрепления
    # немного проще с помощью классов
    root  = Tk()
    title = Button(root, text='PyMail - a Python/Tkinter email client')
    title.config(bg='steelblue', fg='white', relief=RIDGE)
    title.config(command=showhelp)
    title.pack(fill=X)
    decorate(root)
    return root

if __name__ == '__main__':
    # выполнять автономно или прикрепить
    rootWin = makemainwindow(container())    # или makemainwindow()
    rootWin.mainloop
```

Другие инструменты, используемые на стороне клиента

До настоящего момента нас интересовали в этой главе средства Python для работы с FTP и e-mail, и попутно мы познакомились с рядом используемых на стороне клиента модулей: `ftplib`, `poplib`, `smtplib`, `mhlib`, `mimetools`, `urllib`, `rfc822` и т. д. Этот набор хорошо представляет библиотечные средства Python для передачи и обработки информации в Интернете, но он далеко не полон. Более или менее полный список модулей Python, связанных с Интернетом, находится в начале предыдущей главы. Среди прочего Python содержит вспомогательные библиотеки для поддержки на стороне клиента телеконференций Интернета, telnet, HTTP и других стандартных протоколов.

NNTP: доступ к телеконференциям

Модуль Python `nntplib` поддерживает интерфейс клиента к *NNTP* (Network News Transfer Protocol), используемый для чтения и передачи статей в телеконференции Usenet в Интернете. Как и другие протоколы, NNTP выполняется поверх сокетов и просто определяет стандартный протокол сообщений; как и другие модули, `nntplib` скрывает большую часть деталей протокола и предоставляет сценариям Python объектно-ориентированный интерфейс.

Мы не станем здесь вдаваться в детали протокола, но кратко отметим, что серверы NNTP хранят на машине ряд статей, обычно в плоском файле базы данных. Если знать доменное имя или IP-адрес машины, на которой выполняется программа NNTP-сервера, слушающая на NNTP-порту, то можно написать сценарии, получающие или передающие статьи с любой машины, на которой есть Python и соединение с Интернетом. Например, сценарий примера 11.24 по умолчанию получает и отображает первые 10 статей из телеконференции Python *comp.lang.python* с сервера NNTP *news.rmi.net* у моего Интернет-провайдера.

Пример 11.24. PP2E\Internet\Other\readnews.py

```
##########################################################################################
# Получить и вывести сообщения телеконференции usenet comp.lang.python
# с помощью модуля nntplib, выполняемого поверх сокетов; nntplib поддерживает
# также отправку новых сообщений и т. д.; примечание: сообщения не удаляются после прочтения;
##########################################################################################

listonly = 0
showhdrs = ['From', 'Subject', 'Date', 'Newsgroups', 'Lines']
try:
    import sys
    servername, groupname, showcount = sys.argv[1:]
    showcount  = int(showcount)
except:
    servername = 'news.rmi.net'
    groupname  = 'comp.lang.python'   # аргументы командной строки или значения по умолчанию
    showcount  = 10                   # показать последние showcount сообщений

# соединение с сервером nntp
print 'Connecting to', servername, 'for', groupname
from nntplib import NNTP
connection = NNTP(servername)
(reply, count, first, last, name) = connection.group(groupname)
print '%s has %s articles: %s-%s' % (name, count, first, last)

# получить только заголовки для запроса
```

```
fetchfrom = str(int(last) - (showcount-1))
(reply, subjects) = connection.xhdr('subject', (fetchfrom + '-' + last))

# показать заголовки, получить заголовки+тело
for (id, subj) in subjects:                    # [-showcount:] для загрузки всех заголовков
    print 'Article %s [%s]' % (id, subj)
    if not listonly and raw_input('=> Display?') in ['y', 'Y']:
        reply, num, tid, list = connection.head(id)
        for line in list:
            for prefix in showhdrs:
                if line[:len(prefix)] == prefix:
                    print line[:80]; break
        if raw_input('=> Show body?') in ['y', 'Y']:
            reply, num, tid, list = connection.body(id)
            for line in list:
                print line[:80]
    print
print connection.quit()
```

Как и в инструментах FTP и e-mail, этот сценарий создает объект NNTP и вызывает его методы для получения информации телеконференции и заголовков и текста тела статей. Например, метод `xhdr` загружает выбранные заголовки из указанного диапазона сообщений. При запуске эта программа соединяется с сервером и выводит строку темы каждой статьи, останавливаясь для запроса, следует ли получить и показать информационные строки заголовка статьи (только заголовки, перечисленные в переменной `showhdrs`) и текст тела:

```
C:\...\PP2E\Internet\Other>python readnews.py
Connecting to news.rmi.net for comp.lang.python
comp.lang.python has 3376 articles: 30054-33447
Article 33438 [Embedding? file_input and eval_input]
=> Display?

Article 33439 [Embedding? file_input and eval_input]
=> Display?y
From: James Spears <jimsp@ichips.intel.com>
Newsgroups: comp.lang.python
Subject: Embedding? file_input and eval_input
Date: Fri, 11 Aug 2000 10:55:39 -0700
Lines: 34
=> Show body?

Article 33440 [Embedding? file_input and eval_input]
=> Display?

Article 33441 [Embedding? file_input and eval_input]
=> Display?

Article 33442 [Embedding? file_input and eval_input]
=> Display?

Article 33443 [Re: PYHTONPATH]
=> Display?y
Subject: Re: PYHTONPATH
Lines: 13
From: spOOfd <spOOfdNOspSPAM@yahoo.com.invalid>
Newsgroups: comp.lang.python
Date: Fri, 11 Aug 2000 11:06:23 -0700
=> Show body?y
Is this not what you were looking for?
```

```
Add to cgi script:
import sys
sys.path.insert(0, "/path/to/dir")
import yourmodule

--------------------------------------------------------------
Got questions?  Get answers over the phone at Keen.com.
Up to 100 minutes free!
http://www.keen.com

Article 33444 [Loading new code...]
=> Display?

Article 33445 [Re: PYHTONPATH]
=> Display?

Article 33446 [Re: Compile snags on AIX & IRIX]
=> Display?

Article 33447 [RE: string.replace() can't replace newline characters???]
=> Display?

205 GoodBye
```

Можно также использовать этот сценарий другим образом, явно передав ему в командной строке имя сервера, название телеконференции и количество отображаемых сообщений. Вот пример использования этого сценария Python для получения нескольких последних сообщений в телеконференциях Perl и Linux:

```
C:\...\PP2E\Internet\Other>python readnews.py news.rmi.net comp.lang.perl.misc 5
Connecting to news.rmi.net for comp.lang.perl.misc
comp.lang.perl.misc has 5839 articles: 75543-81512
Article 81508 [Re: Simple Argument Passing Question]
=> Display?

Article 81509 [Re: How to Access a hash value?]
=> Display?

Article 81510 [Re: London =?iso-8859-1?Q?=A330-35K?= Perl Programmers Required]
=> Display?

Article 81511 [Re: ODBC question]
=> Display?

Article 81512 [Re: ODBC question]
=> Display?

205 GoodBye

C:\...\PP2E\Internet\Other>python readnews.py news.rmi.net comp.os.linux 4
Connecting to news.rmi.net for comp.os.linux
comp.os.linux has 526 articles: 9015-9606
Article 9603 [Re: Simple question about CD-Writing for Linux]
=> Display?

Article 9604 [Re: How to start the ftp?]
=> Display?

Article 9605 [Re: large file support]
=> Display?

Article 9606 [Re: large file support]
=> Display?y
From: andy@physast.uga.edu (Andreas Schweitzer)
Newsgroups: comp.os.linux.questions,comp.os.linux.admin,comp.os.linux
```

```
Subject: Re: large file support
Date: 11 Aug 2000 18:32:12 GMT
Lines: 19
=> Show body?n

205 GoodBye
```

Потрудившись еще немного, можно было бы превратить этот сценарий в полноценный интерфейс новостей. Например, можно было бы посылать из сценария Python новые статьи с помощью кода такого вида (предполагается присутствие в локальном файле надлежащих строк заголовков NNTP):

```
# для отправки выполните следующее (но только если действительно хотите отправить сообщение!)
connection = NNTP(servername)
localfile = open('filename')      # в файле содержатся правильные заголовки
connection.post(localfile)        # послать текст в телеконференцию
connection.quit()
```

Можно также добавить к этому сценарию GUI на основе Tkinter, чтобы облегчить работу с ним, но это расширение мы добавим к списку упражнений, предлагаемых читателю (см. также в предыдущем разделе предлагаемые расширения к интерфейсу PyMailGui).

HTTP: доступ к сайтам

Стандартная библиотека Python (то есть модули, устанавливаемые вместе с интерпретатором) содержит также поддержку HTTP (Hypertext Transfer Protocol) на стороне клиента – стандарта структуры сообщений и портов, используемых для передачи информации в World Wide Web. Вкратце, это тот протокол, который использует ваш веб-броузер (например, Internet Explorer, Netscape) для получения веб-страниц и запуска приложений на удаленных серверах при веб-серфинге. На нижнем уровне он состоит просто в пересылке байтов через порт 80.

Чтобы действительно понять, как передаются данные по HTTP, необходимо знать некоторые темы, относящиеся к выполнению сценариев на стороне сервера, о чем рассказывается в следующих трех главах (например, как вызываются сценарии и схемы адресации, используемые в Интернете), поэтому данный раздел может оказаться менее полезным для читателей, не имеющих соответствующей подготовки. К счастью, основные интерфейсы HTTP в Python достаточно просты для поверхностного понимания их даже на данном этапе, поэтому мы сейчас кратко их рассмотрим.

Стандартный модуль Python `httplib` в значительной мере автоматизирует использование HTTP и позволяет сценариям получать веб-страницы, почти как в веб-броузерах. В частности, сценарий примера 11.25 может получить любой файл с любой машины, на которой выполняется программа веб-сервера HTTP. Как обычно, файл (и строки заголовков описания) в конечном счете предаются через стандартный порт сокета, но большая часть сложных деталей скрыта в модуле `httplib`.

Пример 11.25. PP2E\Internet\Other\http-getfile.py

```
#########################################################################################
# Получить файл с сервера http (web) через сокеты с помощью httplib;
# параметр имени файла может содержать полный путь к каталогу и указывать
# в конце сценарий cgi с параметрами запроса для запуска удаленной программы;
# данные полученного файла или вывод удаленной программы можно сохранить
# в локальном файле, имитируя ftp, либо разобрать с помощью string.find или модуля htmllib;
#########################################################################################

import sys, httplib
```

```
showlines = 6
try:
    servername, filename = sys.argv[1:]           # аргументы командной строки?
except:
    servername, filename = 'starship.python.net', '/index.html'

print servername, filename
server = httplib.HTTP(servername)                 # соединение с сайтом/сервером http
server.putrequest('GET', filename)                # отправка запроса и заголовков
server.putheader('Accept', 'text/html')           # запросы POST тоже действуют здесь
server.endheaders()                               # как и имена файлов сценариев cgi

errcode, errmsh, replyheader = server.getreply()  # чтение информационных заголовков ответов
if errcode != 200:                                # 200 означает успех
    print 'Error sending request', errcode
else:
    file = server.getfile()                       # файловый объект для полученных данных
    data = file.readlines()
    file.close()                                  # показать строки с eoln в конце
    for line in data[:showlines]: print line,     # для сохранения записать данные в файл
```

Требуемые имена серверов и файлов можно передать в командной строке, переопределив тем самым значения по умолчанию, прошитые в коде. Чтобы лучше понять этот код, необходимо некоторое знание протокола HTTP, но расшифровать его довольно просто. При выполнении клиентом этот сценарий создает объект HTTP, который соединяется с сервером, посылает запрос GET и допустимые типы ответов, а затем читает ответ сервера. Подобно исходному тесту сообщения e-mail, ответ сервера HTTP обычно начинается с нескольких строк заголовков описания, за которыми следует содержимое запрошенного файла. Метод `getfile` объекта HTTP возвращает файловый объект, из которого можно считать загруженные данные.

Давайте получим с помощью этого сценария несколько файлов. Как и все сценарии Python, выполняемые на стороне клиента, данный сценарий работает на любой машине, где есть Python и соединение с Интернетом (в данном случае он выполняется на клиенте Windows). Если все пройдет хорошо, будут выведены несколько первых строк загруженного файла. В более реалистичном приложении получаемый текст следовало бы сохранить в локальном файле, разобрать с помощью модуля Python `htmllib` и т. д. Если не задать аргументы, сценарий просто загрузит страницу оглавления HTML с *http://starship.python.org*:

```
C:\...\PP2E\Internet\Other>python http-getfile.py
starship.python.net /index.html
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen">
  <TITLE>Starship Python</TITLE>
  <SCRIPT language="JavaScript">
<!-- // mask from the infidel
```

Но можно и точно указать в командной строке сервер и файл, который должен быть получен. В следующем коде показано применение этого сценария для получения файлов с двух разных сайтов, имена которых указываются в командных строках (я добавил переносы строк, чтобы уместить их на ширине страницы). Обратите внимание, что аргумент имени файла может содержать произвольный путь к удаленному каталогу с нужным файлом, как в последней из приведенных загрузок:

```
C:\...\PP2E\Internet\Other>python http-getfile.py www.python.org /index.html
www.python.org /index.html
<HTML>
```

```
<!-- THIS PAGE IS AUTOMATICALLY GENERATED.  DO NOT EDIT. -->
<!-- Wed Aug 23 17:29:24 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:

C:\...\PP2E\Internet\Other>python http-getfile.py www.python.org /index
www.python.org /index
Error sending request 404

C:\...\PP2E\Internet\Other>python http-getfile.py starship.python.net  /~lutz/index.html
starship.python.net /~lutz/index.html
<HTML>
<HEAD><TITLE>Mark Lutz's Starship page</TITLE></HEAD>
<BODY>

<H1>Greetings</H1>
```

Обратите также внимание на повторную попытку в этом коде: при неудаче запроса сценарий получает и выводит код ошибки HTTP, возвращаемый сервером (мы забыли задать *.html* в имени файла). При использовании чистого интерфейса HTTP необходимо точно указывать, что требуется получить.

Технически строка в сценарии с именем `filename` может ссылаться на простой статический файл веб-страницы или на программу сервера, генерирующую HTML в качестве вывода. Такие программы, выполняемые на стороне сервера, обычно называются сценариями CGI – они составляют тему следующих трех глав. Пока лишь запомните, что если `filename` указывает на сценарий, то эта программа может запустить другую программу, находящуюся на удаленном сервере. В таком случае можно также указать после `?` параметры (называемые строкой запроса), которые должны быть переданы удаленной программе. В данном примере мы передаем параметр `language=Python` сценарию CGI, с которым мы познакомимся в следующей главе:

```
C:\...\PP2E\Internet\Other>python http-getfile.py starship.python.net
                              /~lutz/Basics/languages.cgi?language=Python
starship.python.net /~lutz/Basics/languages.cgi?language=Python
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Python</H3><P><PRE>
 print 'Hello World'
</PRE></P><BR>
<HR>
```

В этой книге еще много будет говориться о HTML, сценариях CGI и смысле запроса HTTP GET (одного из способов форматирования информации, посылаемой серверу HTTP), поэтому сейчас мы опустим дополнительные детали. Достаточно сказать, однако, что с помощью интерфейсов HTTP можно написать собственный веб-броузер и создать сценарии, использующие сайты так, как если бы они были подпрограммами. Путем отправки параметров удаленным программам и анализа возвращаемых результатов можно заставить сайты играть роль простых функций, выполняющихся в том же процессе (хотя и значительно более медленно и косвенно).

Возвращение к urllib

Модуль `httplib`, с которым мы только что познакомились, предоставляет клиентам HTTP контроль низкого уровня. Однако при работе с объектами, находящимися в Сети, часто оказывается проще написать код загрузки с помощью стандартного модуля Python `urllib`, который был представлен в разделе этой главы, посвященном FTP.

Так как этот модуль дает еще один способ разговаривать на HTTP, остановимся здесь на его интерфейсах.

Вспомните, что при задании URL `urllib` либо загружает запрашиваемый объект из Сети в локальный файл, либо создает файловый объект, который позволяет осуществлять чтение его содержимого. Благодаря этому сценарий примера 11.26 выполняет ту же работу, что и сценарий с использованием `httplib`, который мы только что написали, но требует значительно меньше ввода с клавиатуры.

Пример 11.26. PP2E\Internet\Other\http-getfile-urllib1.py

```
#########################################################################################
# Получить файл с сервера http (web) через сокеты с помощью urllib; urllib поддерживает http,
# ftp, файлы и т.д. через адресные строки url; для hhtp можно задать в url файл или сценарий cgi
# для удаленного запуска; см. также пример urllib в разделе ftp и вызов сценария cgi
# в главе ниже; Python позволяет получать из сети файлы многими способами, различающимися
# сложностью и требованиями к серверам: сокеты, ftp, http, urllib, вывод cgi;
# предостережение: необходимо выполнить urllib.quote над именем файла - см. следующие главы;
#########################################################################################
import sys, urllib
showlines = 6
try:
    servername, filename = sys.argv[1:]                 # аргументы командной строки?
except:
    servername, filename = 'starship.python.net', '/index.html'
remoteaddr = 'http://%s%s' % (servername, filename)     # можно указать сценарий cgi
print remoteaddr
remotefile = urllib.urlopen(remoteaddr)                 # возвращает файловый объект для ввода
remotedata = remotefile.readlines()                     # непосредственное чтение данных
remotefile.close()
for line in remotedata[:showlines]: print line,
```

Почти все детали пересылки по HTTP скрыты здесь за интерфейсом `urllib`. Данная версия работает примерно так же, как версия с `httplib`, которую мы написали сначала, но должна построить и передать URL-адрес Интернета (созданный URL выводится сценарием первой строкой). Как отмечалось в разделе этой главы об FTP, функция `urlopen` модуля `urllib` возвращает объект типа файла, из которого можно читать удаленные данные. Но поскольку создаваемый URL начинается здесь с «http://», модуль `urllib` автоматически применяет для загрузки запрашиваемого файла не FTP, а интерфейсы HTTP низкого уровня:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py
http://starship.python.net/index.html
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen">
  <TITLE>Starship Python</TITLE>
  <SCRIPT language="JavaScript">
<!-- // mask from the infidel

C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py www.python.org /index
http://www.python.org/index
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED.  DO NOT EDIT. -->
<!-- Fri Mar  3 10:28:30 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:
```

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py starship.python.net  /~lutz/index.html
http://starship.python.net/~lutz/index.html
<HTML>
<HEAD><TITLE>Mark Lutz's Starship page</TITLE></HEAD>
<BODY>

<H1>Greetings</H1>

C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py starship.python.net
                                /~lutz/Basics/languages.cgi?language=Java
http://starship.python.net/~lutz/Basics/languages.cgi?language=Java
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Java</H3><P><PRE>
 System.out.println("Hello World");
</PRE></P><BR>
<HR>
```

Как и раньше, аргумент с именем файла может задавать как простой файл, так и вызываемую программу с дополнительными параметрами. Если внимательно изучить этот вывод, то можно заметить, что сценарий работает, даже если опустить *.html* в конце имени файла (во второй командной строке); в отличие от версии чистого HTTP, интерфейс на основе URL достаточно сообразителен, чтобы делать правильные вещи.

Другие интерфейсы urllib

Еще одно последнее видоизменение: в следующем сценарии-загрузчике на базе `urllib` используется интерфейс этого модуля `urlretrieve` несколько более высокого уровня, автоматически сохраняющий загруженный файл или вывод сценария в локальном файле на машине клиента. Этот интерфейс удобен, если действительно нужно сохранить полученные данные (например, чтобы имитировать протокол FTP). Однако если планируется немедленная обработка загружаемых данных, такая форма может оказаться менее удобной, чем только что рассмотренная версия: потребуется открыть и прочесть сохраненный файл. Кроме того, требуется дополнительный протокол для задания и получения имен локальных файлов, как в примере 11.27.

Пример 11.27. PP2E\Internet\Other\http-getfile-urllib2.py

```
#########################################################################################
# Получить файл с сервера http (web) через сокеты с помощью urllib; в данной версии
# используется интерфейс, сохраняющий полученные данные в локальном файле; имя локального файла
# передается в аргументе командной строки или выделяется из url посредством urlparse: аргумент
# имени файла может содержать путь к каталогу в начале и параметры запроса в конце, поэтому
# os.path.split недостаточно (отделяет только путь к каталогу);
# предостережение: нужно выполнить urllib.quote над именем файла - см. следующие главы;
#########################################################################################
import sys, os, urllib, urlparse
showlines = 6
try:
    servername, filename = sys.argv[1:3]                # первые 2 аргумента командной строки?
except:
    servername, filename = 'starship.python.net', '/index.html'

remoteaddr = 'http://%s%s' % (servername, filename)     # любой сетевой адрес
if len(sys.argv) == 4:                                  # получить имя результирующего файла
    localname = sys.argv[3]
else:
    (scheme, server, path, parms, query, frag) = urlparse.urlparse(remoteaddr)
```

```
    localname = os.path.split(path)[1]
print remoteaddr, localname
urllib.urlretrieve(remoteaddr, localname)                # может быть файлом или сценарием
remotedata = open(localname).readlines()                 # сохраняется в локальном файле
for line in remotedata[:showlines]: print line,
```

Запустим этот последний вариант из командной строки. Основные действия такие же, как в последних двух версиях: как в предыдущей версии, строится URL, и, как в обеих последних, можно явно задать целевой сервер и путь к файлу в командной строке:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py
http://starship.python.net/index.html index.html
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen">
  <TITLE>Starship Python</TITLE>
  <SCRIPT language="JavaScript">
<!-- // mask from the infidel

C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py www.python.org /index.html
http://www.python.org/index.html index.html
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED.  DO NOT EDIT. -->
<!-- Wed Aug 23 17:29:24 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:
```

Из-за использования в этой версии интерфейса `urllib`, автоматически сохраняющего загруженные данные в локальном файле, она по духу ближе к загрузке по FTP. Но этот сценарий должен также каким-то образом получить имя локального файла, в который будут записаны данные. Можно позволить сценарию вырезать из созданного URL базовое имя файла, которое и использовать, либо передать имя локального файла в качестве последнего аргумента командной строки. В предшествующем прогоне, к примеру, загруженная веб-страница записывается в локальный файл *index.html* – базовое имя файла, выделенное из URL (сценарий выводит URL и имя локального файла в первой строке вывода). В следующем прогоне имя локального файла передается явным образом как *python-org-index.html*:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py www.python.org
                                        /index.html python-org-index.html
http://www.python.org/index.html python-org-index.html
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED.  DO NOT EDIT. -->
<!-- Wed Aug 23 17:29:24 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:

C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
                                        /~lutz/home/index.html
http://starship.python.net/~lutz/home/index.html index.html
<HTML>

<HEAD>
<TITLE>Mark Lutz's Home Page</TITLE>
</HEAD>

C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
                                        /~lutz/home/about-pp.html
```

```
http://starship.python.net/~lutz/home/about-pp.html about-pp.html
<HTML>

<HEAD>
<TITLE>About "Programming Python"</TITLE>
</HEAD>
```

Ниже приводится листинг, показывающий, как с помощью этой третьей версии запускается удаленная программа. Как и прежде, если не задать имя локального файла явным образом, сценарий выделяет базовое имя файла из аргумента с именем файла. Это не всегда просто или уместно при запуске программ – имя файла может содержать путь к удаленному каталогу в начале и параметры, необходимые для запуска удаленной программы, в конце.

Если задан URL для запуска сценария и не указано явно имя выходного файла, этот сценарий извлекает содержащееся в середине базовое имя файла, применяя сначала стандартный модуль `urlparse` для получения пути к файлу, а затем `os.path.split` для отделения пути к каталогу. Однако в результате получается имя удаленного сценария, которое может оказаться непригодным для локального сохранения данных. Например, в первом прогоне вывод сценария попадает в локальный файл с именем *languages.cgi*, полученным как имя сценария в середине URL; во втором прогоне имя файла указано явно как *CxxSyntax.html*, что подавляет извлечение имени файла из URL:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
                                /~lutz/Basics/languages.cgi?language=Perl
http://starship.python.net/~lutz/Basics/languages.cgi?language=Perl languages.cgi
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Perl</H3><P><PRE>
 print "Hello World\n";
</PRE></P><BR>
<HR>

C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
                           /~lutz/Basics/languages.cgi?language=C++ CxxSyntax.html
http://starship.python.net/~lutz/Basics/languages.cgi?language=C++ CxxSyntax.html
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>C  </H3><P><PRE>
Sorry--I don't know that language
</PRE></P><BR>
<HR>
```

Удаленный сценарий возвращает здесь сообщение о неудаче поиска, когда в последней команде ему передается строка «C++». Дело в том, что «+» представляет собой в строках URL специальный символ (означающий пробел), и для надежности оба написанные нами сценария `urllib` должны были бы пропустить строку `filename` через такую штуку, как `urllib.quote` – средство преобразования специальных символов для передачи. Подробно мы будем говорить об этом в следующей главе, поэтому считайте это лишь предварительным рассмотрением. Но чтобы заставить работать этот вызов программы, необходимо использовать в создаваемом URL специальные последовательности; вот как можно сделать это вручную:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py  starship.python.net
                    /~lutz/Basics/languages.cgi?language=C%2b%2b CxxSyntax.html
http://starship.python.net/~lutz/Basics/languages.cgi?language=C%2b%2b CxxSyntax.html
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>C++</H3><P><PRE>
```

```
 cout &lt;&lt; "Hello World" &lt;&lt; endl;
</PRE></P><BR>
<HR>
```

Странные строки «%2b» в этой командной строке не являются чем-то совершенно таинственным: преобразование, требуемое для URL, можно увидеть, запустив вручную стандартные средства Python (данные сценарии должны делать это автоматически для правильной обработки всех возможных случаев):

```
C:\...\PP2E\Internet\Other>python
Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import urllib
>>> urllib.quote('C++')
'C%2b%2b'
```

Опять же, не усердствуйте, пытаясь понять несколько последних команд: мы вернемся к URL и преобразованиях в них в следующей главе, когда будем изучать сценарии Python, выполняемые на сервере. Там я также объясню, почему результат для C++ был возвращен со странными символами `<<` – кодами HTML для <<.

Прочие возможности создания сценариев клиентов

В данной главе мы сосредоточились на интерфейсах клиентской стороны к стандартным протоколам, работающим через сокеты, но программирование для стороны клиента может также принимать другие формы. Например, в главе 15 мы увидим, что код Python можно встраивать внутрь кода HTML, описывающего веб-страницу, с помощью расширения Windows Active Scripting. При загрузке в Internet Explorer файла такой веб-страницы с веб-сервера встроенные сценарии Python фактически выполняются на машине клиента, при этом объектный API предоставляет доступ к контексту броузера. Код HTML первоначально загружается через сокет, но его выполнение не связано с протоколом, основанным на сокетах.

В главе 15 мы также познакомимся с такими средствами для стороны клиента, как система *JPython* (иначе называемая «Jython») – компилятор, поддерживающий написанные на Python *апплеты* Java, представляющие собой программы общего назначения, загружаемые с сервера и выполняемые локально у клиента при обращении к ним или ссылке по URL. Мы также посмотрим на средства Python для обработки *XML* – структурированного текста, который в будущем может стать стандартным языком для диалогов клиент/сервер.

Однако принимая во внимание отсутствие времени и места, мы не станем здесь глубже вникать в детали этих и других средств, используемых на стороне клиента. Если вас интересует применение Python в сценариях клиентов, следует потратить немного времени и ознакомиться с перечнем инструментов Интернета, имеющимся в справочном руководстве по библиотеке Python. Все они действуют на сходных принципах, но имеют несколько различающиеся интерфейсы.

В следующей главе мы перемахнем по ту сторону Интернета и рассмотрим сценарии, выполняющиеся на серверах. Такие программы лежат в основе важного понятия программ, целиком живущих в Веб и запускаемых веб-броузерами. Осуществляя этот скачок в структуре, нужно помнить, что средств, с которыми мы ознакомились в этой и предыдущей главах, часто достаточно для полной реализации распределенной обработки, требуемой во многих приложениях, и работать они могут в согласии со сценариями, выполняемыми на сервере. Однако для полного понимания картины мира Веб необходимо также исследовать и царство серверов.

12

Сценарии, выполняемые на сервере

«До чего же запутанную паутину мы плетем...»

Из этой главы состоит третья часть нашего рассмотрения программирования для Интернета на Python. В предыдущих двух главах мы изучили сокеты и основные интерфейсы программирования клиентов, такие как FTP и e-mail. В этой главе наше внимание будет сосредоточено на создании в Python *сценариев*, *выполняемых на стороне сервера*, – программ, обычно называемых *сценариями CGI*. Сценарии, выполняемые на серверах, и производные от них во многом определяют то, что происходит в Сети в наше время.

Как будет показано, Python служит идеальным языком для создания сценариев, которые реализуют и индивидуально настраивают сайты, благодаря как простоте применения, так и поддержке библиотеками. В следующих двух главах базовые знания, полученные из этой главы, будут использованы для реализации развитых сайтов. После этого мы завершим тему главой, в которой будут рассмотрены другие связанные с Интернетом темы и технологии. Здесь же нашей целью будет понять основы сценариев, выполняемых на стороне сервера, прежде чем исследовать системы, основанные на этой базовой модели.

Что такое сценарий CGI для сервера?

Попросту говоря, сценарии CGI реализуют значительную часть того взаимодействия, с которым вы сталкиваетесь в Веб. Это стандартный и широко используемый способ взаимодействия с сайтами. Существуют другие способы сделать поведение сайтов интерактивным с помощью Python, в том числе решения для стороны клиента (например, апплеты JPython и Active Scripting), а также серверные технологии, основанные на базовой модели CGI (например, Active Server Pages и Zope). Их мы также вкратце обсудим в конце главы 15 «Более сложные темы Интернета». В целом сценарии CGI для серверов применяются в программировании значительной части Веб.

Притаившийся сценарий

Формально говоря, сценарии CGI являются программами, выполняющимися на машине сервера и придерживающимися общего шлюзового интерфейса (Common Gateway Interface) – модели связи между броузером и сервером. Разобраться с CGI, видимо, удобнее исходя из предполагаемого им взаимодействия.

Большинство тех, кто путешествует по Сети и нажимает кнопки на веб-страницах, воспринимает такое взаимодействие как данность, но за кулисами каждой операции в Сети происходит масса вещей. С точки зрения пользователя, это достаточно знакомый и простой процесс:

Замок на песке

При чтении последующих трех глав этой книги имейте, пожалуйста, в виду, что они должны служить лишь введением в создание на Python сценариев для серверов. Область деятельности веб-мастеров велика и сложна, непрерывно меняется и часто предписывает несколько способов достижения заданной цели, некоторые из которых могут разниться от броузера к броузеру и от сервера к серверу. Например, схема шифрования паролей из следующей главы может в некоторых ситуациях оказаться излишней, а особые теги HTML иногда могут помочь избавиться от части работы, которую мы здесь проделываем.

Учитывая, сколь велика и изменчива эта база знаний, данная часть книги даже не претендует на полноту рассмотрения области создания сценариев для серверов. Чтобы стать настоящим специалистом в этой сфере, нужно изучить другие книги, излагающие дополнительные детали и приемы веб-мастерства (например, Ч. Муссиано (Ch. Musciano) и Б. Кеннеди (B. Kennedy) «HTML & XHTML: The Definitive Guide», издательство O'Reilly)[1]. Здесь же вы познакомитесь с инструментальным набором Python для CGI и получите знания, достаточные для того, чтобы начать создавать на Python собственные значительные сайты. Но не следует воспринимать эту книгу как последнее слово по данной теме.

[1] Чак Муссиано и Билл Кеннеди. «HTML и XHTML. Подробное руководство». – СПб: издательство «Символ-Плюс», 2002.

1. *Передача*. Когда вы приходите на сайт, чтобы приобрести товар или передать данные, то обычно заполняете форму в веб-броузере, нажимаете кнопку, чтобы передать информацию, и ждете ответа.
2. *Ответ*. Предполагая, что с вашим соединением с Интернетом и компьютером, с которым вы взаимодействуете, все в порядке, вы в итоге получает ответ в виде новой веб-страницы. Это может быть просто подтверждение (например, «Спасибо за сделанный вами заказ») или новая форма, которую опять нужно заполнить и послать.

И, верьте или нет, эта простая модель лежит в основе всей кипучей жизни в Сети. Но внутри все несколько сложнее. В действительности здесь действует тонкая архитектура клиент/сервер, базирующаяся на сокетах, – веб-броузер, выполняющийся на вашей машине, является *клиентом*, а компьютер, с которым вы связаны через Интернет, является *сервером*. Рассмотрим снова схему взаимодействия со всеми внутренними деталями, обычно невидимыми для пользователей.

Передача

После заполнения страницы с формой в веб-броузере и нажатия кнопки передачи броузер незаметно для вас пересылает информацию через Интернет на машину сервера, указанную в качестве получателя. Машина сервера обычно является удаленным компьютером, расположенным в другом месте как кибернетического, так и реального пространства. Он указан в URL, к которому происходит обращение (строка Интернет-адреса, появляющаяся наверху броузера). Целевые сервер и файл могут быть заданы в явно введенном URL, но чаще они указываются в HTML, описывающем саму страницу для передачи – в гиперссылке или теге «action» формы HTML. Каким бы образом ни был указан сервер, выполняющийся на компьютере броузер в конечном счете посылает ему вашу информацию через сокет с помощью технологий, рассмотренных нами в предыдущих двух главах. На машине сервера постоянно выполняется программа, называемая HTTP-сервером, которая ждет на сокете поступления данных от броузеров, обычно на порту с номером 80.

Обработка

Когда ваша информация оказывается на машине сервера, программа HTTP-сервера сначала должна обнаружить ее и решить, как обработать запрос. Если запрашиваемый URL указывает просто на *веб-страницу* (например, URL, оканчивающийся на *.html*), HTTP-сервер открывает указанный HTML-файл на машине сервера и передает его текст броузеру через сокет. На стороне клиента броузер считывает HTML и строит по нему страницу, которую вы видите. Но если запрашиваемый броузером URL указывает на *выполняемую программу* (например, если URL оканчивается на *.cgi*), HTTP-сервер для обработки запроса запускает ее на машине сервера и переадресует данные, поступающие от броузера, указанной программе на машине сервера, чтобы та обработала запрос, и переадресует полученные от броузера данные во входной поток `stdin` и переменные окружения порожденной программы. Обычно эта программа является CGI-сценарием – программой, выполняемой на удаленной машине сервера где-то в киберпространстве, а не на вашем компьютере. С этого момента за обработку запроса отвечает эта программа: она может сохранить ваши данные в базе, списать средства с вашей кредитной карточки и т. д.

Ответ

В конечном итоге CGI-сценарий выводит HTML, который должен создать в вашем броузере новую страницу ответа. При запуске CGI-сценария HTTP-сервер должен обеспечить соединение стандартного выходного потока сценария `stdout` с сокетом, на котором броузер ждет данных. Благодаря этому код HTML, выводимый CGI-сценарием, передается через Интернет вашему броузеру и создает новую страницу. HTML, выводимый CGI-сценарием, действует точно так же, как если бы он хранился и считывался из файла HTML: он может описывать простую страницу ответа или совершенно новую форму для получения дополнительной информации.

Иными словами, CGI-сценарии представляют собой нечто вроде *обработчиков обратного вызова* для генерируемых веб-броузерами запросов, требующих динамического выполнения программ; они автоматически запускаются на машине сервера в ответ на действия в броузере. Хотя CGI-сценарии получают и посылают стандартные структурированные сообщения через сокеты, CGI более похож на процедурное соглашение более высокого уровня для обмена информацией между броузером и сервером.

Создание сценариев CGI на Python

Если все описанное выше кажется сложным, успокойтесь: Python, а также HTTP-сервер, выполняющий сценарии, автоматизируют большую часть трудных задач. Сценарии CGI пишутся как вполне автономные программы и предполагают, что задачи начального запуска уже решены. Серверная часть протокола HTTP реализуется веб-сервером, а не CGI-сценарием. Кроме того, библиотечные модули Python автоматически препарируют информацию, переданную броузером, и передают ее сценарию CGI в легко усвояемой форме. В результате сценарии могут сосредоточиться на деталях приложения, таких как обработка полученных данных и создание конечной страницы.

Как отмечалось выше, в контексте сценариев CGI потки `stdin` и `stdout` автоматически привязываются к сокетам, соединенным с броузером. Кроме того, HTTP-сервер передает часть информации от броузера CGI-сценарию в виде переменных окружения оболочки. Для программистов CGI это означает:

- *Входные* данные, передаваемые броузером серверу, становятся потоком байтов во входном потоке `stdin`, а также попадают в переменные окружения оболочки.
- *Выходные* данные посылаются сервером клиенту просто в результате вывода надлежащим образом форматированного HTML в выходной поток `stdout`.

Наиболее сложными частями в этой схеме оказываются синтаксический анализ всех входных данных, посылаемых броузером, и форматирование данных в возвращаемом броузеру ответе. К счастью, стандартная библиотека Python в значительной мере автоматизирует обе задачи:

Ввод

С помощью модуля Python `cgi` входные данные, введенные в форму веб-броузера или прикрепленные к строке URL, становятся для CGI-сценариев Python значениями в объекте типа словаря. Python сам анализирует данные и совершенно независимо от стиля передачи (форма или URL) возвращает объект, в котором для каждого данного, посланного броузером, есть одна пара `key:value`.

Вывод

В модуле `cgi` есть также средства для автоматического преобразования строк к виду, допустимому в HTML (например, встроенные символы `<`, `>` и `&` заменяются escape-последовательностями HTML). Модуль `urllib` предоставляет другие средства для форматирования текста, вводимого в генерируемые строки URL (например, добавление escape-последовательностей `%XX` и `+`).

Ниже в этой главе мы подробно изучим оба эти интерфейса. Сейчас же просто имейте в виду, что хотя сценарии CGI можно писать на любом языке, стандартные модули Python и атрибуты языка делают эту задачу очень легкой.

Менее приятно то, что сценарии CGI тесно связаны с синтаксисом HTML, поскольку он должен генерироваться для создания страницы ответа. На самом деле можно сказать, что в CGI-сценарии Python встраивается HTML, являющийся совершенно отдельным самостоятельным языком. Как будет показано, то обстоятельство, что сценарии CGI создают интерфейс пользователя путем вывода синтаксиса HTML, означает необходимость особенно тщательно следить за тем, какой текст будет помещен в код веб-страницы (например, преобразовывать операторы HTML в escape-последовательности). Еще хуже то, что для создания сценариев CGI требуется хотя бы беглое знание форм HTML, поскольку в них обычно и задаются входные данные и адрес целевого сценария. Эта книга не ставит перед собой задачу обучения HTML; если вас озадачит таинственный синтаксис HTML, генерируемый приводимыми здесь сценариями, следует взглянуть на какое-нибудь введение в HTML, например книгу Муссиано и Кеннеди «HTML и XHTML. Подробное руководство».

Запуск примеров сценариев, выполняемых на сервере

Подобно GUI, системы, действующие в веб, весьма интерактивны, и лучшим способом получить практическое представление о некоторых из этих примеров будет опробовать их реально. Прежде чем разбирать какой-либо код, отметим, что для *выполнения* примеров в нескольких следующих главах не потребуется ничего, кроме броузера. Это означает, что все примеры для веб, которые будут здесь показаны, могут выполняться из любого веб-броузера на любой машине независимо от наличия на машине Python. Просто наберите в броузере такой URL:[1]

```
http://starship.python.net/~lutz/PyInternetDemos.html
```

[1] Могут пройти многие годы, прежде чем эта книга будет переиздана, и за это время имя сервера *starship.python.net* в указанном адресе может измениться. Если обращение по этому адресу окажется неудачным, проверьте исправления к данной книге на *http://rmi.net/~lutz/about-pp.html* и посмотрите, не указан ли там новый адрес сайта примеров. Остальные URL главной страницы вряд ли изменятся. Обратите, однако, внимание, что в некоторых примерах имя хоста сервера *starship* прошито в URL; адреса будут исправлены на новые при перемещении сервера, но только не на CD вашей книги. Для автоматического исправления наваний сайтов выполняйте сценарий *fixsitename.py*, приводимый далее в этой главе.

По этому адресу загружается страница для запуска программ со ссылками на все файлы примеров, установленные на машине сервера, доменным именем которого является *starship.python.net* (машина, посвященная разработчикам Python). Сама страница для запуска при загрузке в Internet Explorer выглядит так, как показано на рис. 21.1. В других броузерах она выглядит аналогично. Для каждого крупного примера есть ссылка на этой странице, при щелчке по которой он выполняется.

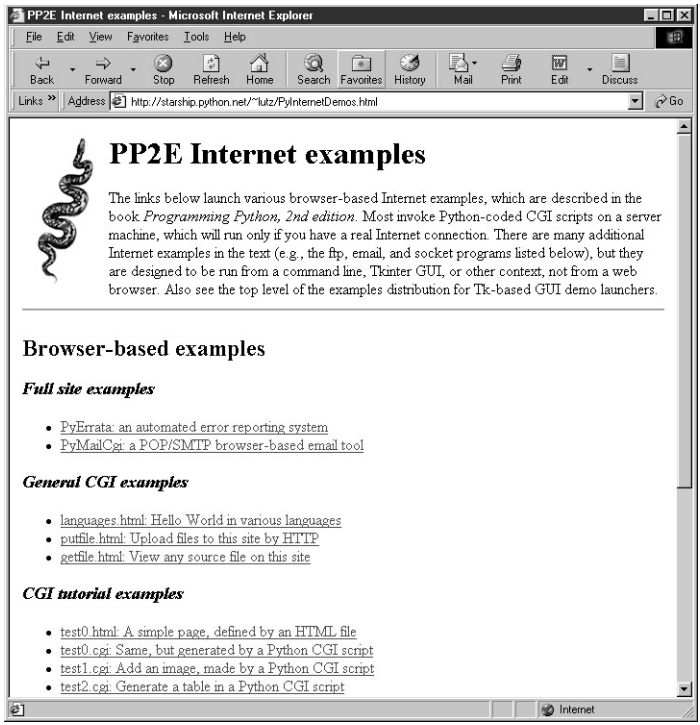


Рис. 12.1. PyInternetDemos – страница запуска сценариев

Страницу запуска сценариев и все файлы HTML из этой главы можно также загрузить локально из каталога дистрибутива примеров данной книги на вашей машине. Можно даже непосредственно открыть их с CD, прилагаемого к книге, и с помощью кнопок программ запуска демонстраций верхнего уровня данной книги. Однако сценарии CGI, вызываемые в конечном счете некоторыми ссылками из примеров, должны выполняться на сервере и потому требуют действующего соединения с Интернетом. Если просматривать корневые страницы локально на машине, то броузер либо будет отображать исходный код сценариев, либо сообщит о необходимости соединиться с Сетью, чтобы запустить сценарий CGI. В Windows диалог для соединения с сетью должен появиться автоматически, когда оно потребуется.

Изменение примеров на сервере

Конечно, выполнять сценарии в броузере – это не совсем то, что самостоятельно писать сценарии. Если вы решите изменить эти программы CGI или написать новые с чистого листа, то должны иметь доступ к машинам веб-серверов:

- Чтобы изменять сценарии на стороне сервера, необходимо иметь учетную запись на машине веб-сервера с установленной версией Python. Стандартной учетной за-

писи часто оказывается достаточно. В этом случае можно редактировать сценарии на своей машине и загружать их на сервер через FTP.

- Чтобы явно вводить командные строки на машине сервера или непосредственно редактировать сценарии на сервере, потребуется также иметь доступ к оболочке на веб-сервере. Такой доступ позволит подключиться к этой машине через telnet и получить приглашение командной строки.

В отличие от примеров из предыдущей главы, для сценариев Python, выполняемых на стороне сервера, требуются как Python, так и сервер. Это означает, что вам потребуется доступ к машине веб-сервера, поддерживающего сценарии CGI в целом, и где уже установлен интерпретатор Python или разрешается установить его самостоятельно. Интернет-провайдеры относятся к этому по-разному, но вариантов может быть несколько, как на коммерческой основе, так и бесплатно (подробнее об этом ниже).

Определив сервер, на котором будут находиться ваши сценарии, можно модифицировать и загрузить по FTP файлы с исходным CGI-кодом с прилагаемого к книге CD на собственный сервер. После этого может потребоваться также выполнить на сервере два сценария Python командной строки, *fixcgi.py* и *fixsitename.py*, представленные далее в этой главе. Первый из них устанавливает права доступа к сценариям CGI, а второй заменяет все упоминания имени сервера *starship* в ссылках и формах, имеющихся в примерах, именем вашего собственного сервера. Другие детали установки сценариев мы изучим ниже в этой главе, а в конце главы 15 рассмотрим некоторые возможности настройки серверов.

Просмотр сценариев, находящихся на сервере, и их вывода

Исходный код примеров, относящихся к данной части книги, приведен в тексте и помещен на прилагаемый CD. В тех случаях, когда нужно увидеть исходный код файла HTML или код HTML, созданный CGI-сценарием Python, можно просто выбрать пункт меню броузера View Source («Просмотр исходного кода» или «Просмотр HTML-кода» в локализованных версиях броузеров) во время отображения соответствующей веб-страницы.

Имейте, однако, в виду, что опция броузера View Source позволяет увидеть вывод сценария сервера после его выполнения, а не исходный код самого сценария. Не существует автоматического способа просмотра исходного кода Python самих CGI-сценариев, кроме поиска их в книге или на ее CD.

Для решения этой проблемы ниже в этой главе мы напишем CGI-программу с именем `getfile`, которая позволяет загрузить и просмотреть исходный код любого файла (HTML, сценарий CGI и т. д.), находящегося на сайте этой книги. Нужно просто ввести имя нужного файла в форму веб-страницы, на которую указывает ссылка *getfile.html* со страницы запуска демонстрационных программ Интернета, или добавить его в качестве параметра в конец явно вводимого URL, например:

```
http://.../getfile.cgi?filename=somefile.cgi
```

В ответ сервер возвратит текст указанного файла в ваш броузер. Этот процесс требует, однако, явного обращения к интерфейсу и значительно больших знаний, чем у нас пока есть, поэтому детали будут объяснены дальше.

Взбираясь по кривой обучения CGI

Пора заняться конкретными деталями программирования. В этом разделе знакомство с написанием кода CGI осуществляется поэтапно – от простых неинтерактивных сценариев до больших программ, использующих все стандартные инструменты веб-

страниц для ввода данных пользователем (те, что были названы графическими элементами («widgets») в главах по GUI Tkinter части II «Программирование GUI»). Сначала мы будем продвигаться неспешно, чтобы полностью изучить основы; в последующих двух главах идеи, с которыми мы здесь познакомимся, будут использованы для создания примеров более крупных и практических сайтов. А пока давайте изучим простой начальный учебник CGI, в котором HTML не больше, чем нужно для написания базовых сценариев для сервера.

Первая веб-страница

Как уже говорилось, сценарии CGI тесно связаны с HTML, поэтому начнем с простой страницы HTML. Файл *test0.html*, показанный в примере 12.1, определяет настоящую полностью действующую веб-страницу. Это текстовый файл, содержащий код HTML, который задает структуру и содержимое простой веб-страницы.

Пример 12.1. PP2E\Internet\Cgi-Web\Basics\test0.html

```
<HTML><BODY>
<TITLE>HTML 101</TITLE>
<H1>A First HTML page</H1>
<P>Hello, HTML World!</P>
</BODY></HTML>
```

Если направить веб-броузер на адрес этого файла в Интернете (или указать локальный путь к нему на вашей машине), то должна появиться страница типа показанной на рис. 12.2. Здесь показано, как выглядит эта страница в Internet Explorer; в других броузерах она выводится аналогично.

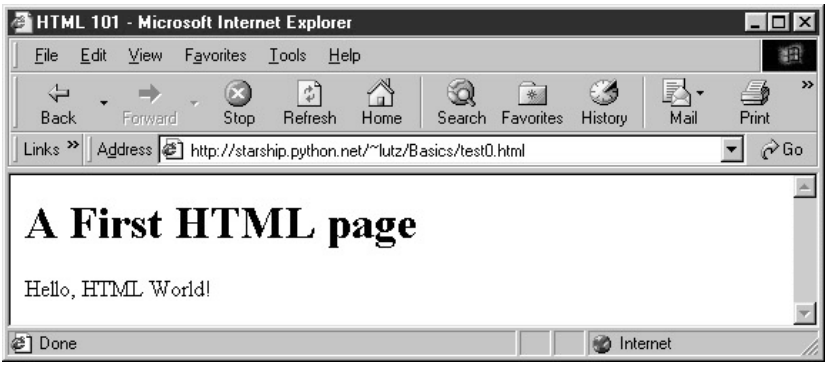


Рис. 12.2. Простая веб-страница из файла HTML

Чтобы действительно разобраться в том, как работает этот маленький файл, требуются некоторые знания о правах доступа, синтаксисе HTML и адресах Интернета. Рассмотрим кратко каждую из названных тем, а потом перейдем к более крупным примерам.

Ограничения прав доступа к файлам HTML

Прежде всего, если нужно установить этот код на другую машину, обычно необходимо предоставить всем пользователям право чтения файлов веб-страниц и их каталогов. Это требуется потому, что их будет загружать из Сети произвольный пользователь (в действительности некто с именем «nobody», с которым мы познакомимся чуть ниже). На Unix-подобных машинах изменение прав доступа может осуществляться соответствующей командой *chmod*. Например, обычно достаточно команды оболочки

`chmod 755 filename`; она дает всем права чтения и выполнения *filename*, а запись разрешается только вам.[1] Такие права доступа к файлам и каталогам являются типичными, но на некоторых серверах могут быть отличия. Загружая этот файл на свой сайт, поинтересуйтесь тем, какие соглашения приняты на вашем сервере.

Основы HTML

Я обещал, что не стану много рассказывать в этой книге об HTML, но для понимания примеров некоторые знания требуется иметь. Вкратце, HTML представляет собой описательный язык разметки, основанный на *тегах* – элементах, заключенных в пару символов <>. Одни теги являются самостоятельными (например, `<HR>` задает горизонтальную линию). Другие действуют парно, обозначая начало и конец, причем в концевой тег входит дополнительный слэш.

Например, чтобы задать текст строки заголовка уровня 1, записывается код HTML вида `<H1>`*text*`</H1>`; текст, находящийся между тегами, выводится на веб-странице. Некоторые теги позволяют также указывать параметры. Например, пара тегов вида `<A href="`*address*`">`*text*`</A>` задает *гиперссылку*: при нажатии на текст ссылки на странице происходит переадресация броузера на Интернет-адрес (URL), указанный в параметре `href`.

Важно помнить, что HTML используется только для описания страниц: веб-броузер читает код HTML и переводит содержащееся в нем описание в веб-страницу, на которой есть заголовки, абзацы, ссылки и т. п. Примечательно, что отсутствуют информация о расположении – за размещение составляющих на странице отвечает броузер, и синтаксис логики программирования – нет операторов «if», циклов и т. д. В этом файле вы также не найдете кода Python; исходный HTML предназначен строго для описания страниц, а не кодирования программ или задания всех деталей интерфейса пользователя.

Отсутствие в HTML управления интерфейсом пользователя и программируемости является одновременной сильной и слабой его чертой. Он хорошо приспособлен для описания страниц и простых интерфейсов пользователя на высоком уровне. Броузер, а не вы, занимается физическим размещением страницы на экране. С другой стороны, HTML непосредственно не поддерживает развитые GUI, что вызывает необходимость применения на сайтах CGI-сценариев (и других технологий) для введения динамической программируемости в статичный по сути HTML.

Интернет-адреса (URL)

Написав файл HTML, вы должны поместить его в таком месте, где он будет доступен окружающему миру. Как и все файлы HTML, *test0.html* должен быть записан в каталог на машине сервера, из которого выполняющаяся на машине программа веб-сервера разрешает брать страницы броузерам. На сервере, где находится этот пример, файл страницы должен храниться в каталоге *public_html* или ниже моего личного исходного каталога, то есть где-то в дереве каталогов, растущем из */home/lutz/public_html*. Примеры для этого раздела находятся в подкаталоге *Basics*, поэтому полным путем в Unix к этому файлу на сервере будет:

```
/home/lutz/public_html/Basics/test0.html
```

[1] Это не какие-то магические числа. На Unix-машинах режим 755 является битовой маской. Первая цифра 7 означает, что вы (владелец файла) можете читать, записывать и выполнять файл (7 в двоичном виде есть 111, и каждый бит разрешает режим доступа). Две цифры 5 (двоичное 101) указывают, что все остальные (ваша группа и те, кто остался) могут осуществлять чтение и выполнение (но не запись) файла. Детали смотрите в системной странице руководства по команде *chmod*.

Этот путь отличается от адреса *PP2E\Internet\Cgi-Web\Basics* на прилагаемом к книге CD, указанного в заголовке листинга файла примера. Однако клиент при ссылке на этот файл должен указывать его адрес в Интернете, который иногда называют *URL*. Чтобы загрузить удаленную страницу, введите в поле адреса броузера следующий текст (или щелкните по гиперссылке *test0.html* на корневой странице примеров, которая указывает на тот же адрес):

```
http://starship.python.net/~lutz/Basics/test0.html
```

Эта строка представляет собой URL, составленный из нескольких частей:

Имя протокола: http

Часть этого URL, содержащая протокол, сообщает броузеру, что он должен связаться с программой HTTP-сервера на машине сервера с помощью протокола сообщений HTTP. В URL, используемых в броузерах, могут указываться различные протоколы – например, *ftp://* для ссылки на файл, управляемый протоколом и сервером FTP, *telnet* для начала сеанса клиента telnet, и т. д.

Имя машины сервера: starship.python.net

Вслед за типом протокола URL указывает также машину целевого сервера. В данном случае это доменное имя машины сервера, где установлены примеры. Заданное имя используется для открытия сокета для связи с сервером. В случае HTTP сокет обычно соединяется с портом номер 80.

Путь к файлу: ~lutz/Basics/test0.html

Наконец, URL задает путь к нужному файлу на удаленной машине. Веб-сервер HTTP автоматически транслирует путь к файлу из URL в действительный путь к файлу в Unix: на моем сервере *~lutz* автоматически транслируется в каталог *public_html* в моем исходном каталоге. Обычно URL отображаются в такие файлы, но могут ссылаться и на другие виды элементов.

Параметры (есть в более поздних примерах)

За URL могут также следовать дополнительные входные параметры для программ CGI. Параметры указываются после знака `?` и отделяются один от другого символом `&`; например, строка вида `?name=bob&job=hacker` в конце URL передает параметры `name` и `job` CGI-сценарию, ранее указанному в URL. Эти значения иногда называются параметрами строки запроса URL и обрабатываются так же, как данные, вводимые из формы. Подробнее о формах и параметрах ниже.

Для полноты следует также знать, что в URL может содержаться дополнительная информация (например, в части, указывающей имя сервера, может также через `:` задаваться номер порта), но здесь мы не станем касаться этих дополнительных правил форматирования. Если вас интересуют дополнительные подробности, можно начать с чтения описания модуля `urlparse` в руководстве по библиотеке Python, а также его исходного кода в стандартной библиотеке Python. Можно также обратить внимание, что URL, который вводится для доступа к некоторой странице, выглядит несколько иначе после того, как страница получена (пробелы превращаются в символы `+`, добавляются `%` и т. д.). Это связано с тем, что броузеры, как правило, должны следовать соглашениям по преобразованию (то есть трансляции) URL, которые будут изучены далее в этой главе.

Использование минимальных URL

Поскольку броузеры запоминают Интернет-адрес предыдущей страницы, URL, встроенные в HTML-файлы, часто могут опускать имена протоколов и серверов, а также путь к каталогу файла. При отсутствии каких-то частей броузер просто использует их значения, взятые из адреса предшествующей страницы. Такой минимальный син-

таксис действует как для URL, встроенных в гиперссылки, так и для действий форм (с формами мы познакомимся ниже). Например, внутри страницы, полученной из каталога *dirpath* на сервере *www.server.com*, такие минимальные гиперссылки и действия форм, как

```
<A HREF="more.html">
<FORM ACTION="next.cgi" ...>
```

обрабатываются в точности так же, как если бы был задан полный URL с явными составными частями сервера и пути, как в следующем коде:

```
<A HREF="http://www.server.com/dirpath/more.html">
<FORM ACTION="http://www.server.com/dirpath/next.cgi" ...>
```

Первый минимальный URL ссылается на файл *more.html* на том же самом сервере и в том же самом каталоге, откуда получена страница, содержащая эту гиперссылку; броузер расширяет его до полного URL. В составляющей пути файла URL может также применяться синтаксис относительного пути в стиле Unix. Например, тег гиперссылки типа `<A HREF="../spam.gif">` задает GIF-файл в родительском каталоге файла, содержащего URL этой ссылки на той же машине сервера.

К чему нужна вся эта возня с укороченными URL? Помимо увеличения срока службы клавиатуры и сохранения зрения главным преимуществом таких минимальных URL является то, что их не понадобится изменять при перемещении страниц в новый каталог или сервер – сервер и путь логически определяются при использовании страницы, а не кодируются жестко в ее HTML. В противном случае последствия могут оказаться достаточно болезненными: примеры, содержащие явные ссылки на сайты и имена путей в URL, находящихся в коде HTML, нельзя копировать на другие серверы без внесения изменений в исходный код. Помощь в этом могут оказать специальные сценарии, но редактирование исходного кода чревато возникновением ошибок.[1]

Недостатком минимальных URL является то, что при переходе по ним не происходит автоматического соединения с Интернетом. Это становится заметным только при загрузке страниц из локальных файлов на вашем компьютере. Например, обычно можно открывать страницы HTML вообще без соединения с Интернетом путем направления веб-броузера на файл страницы, расположенный на локальной машине (скажем, щелкнув по значку этого файла). При таком локальном просмотре страницы переход по полностью заданному URL заставляет броузер автоматически соединиться с Интернетом, чтобы получить нужную страницу или сценарий. Однако минимальные URL открываются снова на локальной машине; обычно броузер просто выводит исходный код страницы или сценария, на которые приведена ссылка.

В итоге оказывается, что минимальные URL лучше переносимы, но успешнее работают, когда все страницы реально загружаются из Интернета. Для облегчения работы с примерами, приведенными в этой книге, в содержащихся в них URL часто опускаются составляющие с именем сервера и путем. Для данной книги, чтобы получить истинный URL страницы или сценария из минимального, представьте себе, что строка

```
http://starship.python.net/~lutz/subdir
```

находится перед именем файла, заданным в URL. В любом случае, даже если вы этого не сможете сделать, это сделает ваш броузер.

---

[1] Для облегчения этого процесса может быть использован представленный в следующем разделе сценарий *fixsitename.py*, который существенно автоматизирует внесение необходимых изменений, выполняя операции глобального поиска и замены и обходя каталоги. В книге есть несколько примеров, использующих полные URL, поэтому после копирования примеров на новый сайт не забудьте выполнить этот сценарий.

Первый CGI-сценарий

HTML-файл, который мы только что видели, и является HTML-файлом, а не сценарием CGI. При обращении броузера к нему удаленный веб-сервер просто отправляет обратно текст файла, с помощью которого в броузере создается новая страница. Чтобы проиллюстрировать природу CGI-сценариев, перепишем этот пример в виде CGI-программы Python, как показано в примере 12.2.

Пример 12.2. PP2E\Internet\Cgi-Web\Basics\test0.cgi

```
#!/usr/bin/python
##################################################################################################
# Выполняется на сервере, выводит html для создания новой страницы; права доступа на исполнение,
# хранится в ~lutz/public_html,  url=http://starship.python.net/~lutz/Basics/test0.cgi
##################################################################################################

print "Content-type: text/html\n"
print "<TITLE>CGI 101</TITLE>"
print "<H1>A First CGI script</H1>"
print "<P>Hello, CGI World!</P>"
```

Этот файл, *test0.cgi*, создаст такую же страницу, если направить на него броузер (просто замените в URL *.html* на *.cgi*). Но это совсем другой зверь – это выполняемая программа, запускаемая на сервере в ответ на запрос доступа. Это также совершенно законная программа Python, в которой HTML страницы выводится динамически, а не заготовлен заранее в статичном файле. На самом деле в этой программе Python вообще мало чего специфического для CGI; при запуске из командной строки она просто выводит HTML, а не генерирует страницу броузера:

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python test0.cgi
Content-type: text/html

<TITLE>CGI 101</TITLE>
<H1>A First CGI script</H1>
<P>Hello, CGI World!</P>
```

Однако при выполнении этой программы HTTP-сервером на машине веб-сервера стандартный выходной поток привязывается к сокету, из которого осуществляет чтение броузер на машине клиента. В таком контексте весь вывод пересылается через Интернет в броузер и должен быть отформатирован в соответствии с ожиданиями броузера. В частности, когда вывод сценария достигает броузера, первая выведенная строка интерпретируется как заголовок, описывающий последующий текст. В выводимом ответе может быть более одной строки заголовка, но между заголовками и началом кода HTML (или другими данными) всегда должна быть одна пустая строка.

В данном сценарии первая строка заголовка сообщает броузеру, что дальнейшая передача представляет собой текст HTML (*text/html*), а символ новой строки (`\n`) в конце первого оператора `print` генерирует дополнительный перевод строки помимо того, который создает сам оператор `print`. Оставшаяся часть вывода программы представляет собой стандартный HTML и используется броузером для генерации у клиента веб-страницы, в точности так, как если бы HTML находился в статическом HTML-файле на сервере.[1]

---

[1] Обратите внимание, что этот сценарий не генерирует охватывающие теги `<HEAD>` и `<BODY>`, имеющиеся в статическом файле HTML из предыдущего раздела. Строго говоря, он должен был бы сделать это – HTML без таких тегов является недействительным. Но все стандартные броузеры не обращают внимания на такое упущение.

Обращение к сценариям CGI производится в точности, как к файлам HTML: либо полный URL этого сценария вводится в поле адреса броузера, либо производится щелчок по строке ссылки *test0.cgi* в корневой странице примеров (осуществляющий переход по минимальной гиперссылке, разрешаемой в полный URL сценария). На рис.12.3 показана результирующая страница, которая будет сгенерирована, если направить броузер на этот сценарий и заставить его выполниться.

Рис. 12.3. Простая веб-страница, создаваемая сценарием CGI

Установка сценариев CGI

Как и файлы HTML, сценарии CGI являются простыми текстовыми файлами, которые можно создать на локальной машине и загрузить на сервер по FTP либо написать в тестовом редакторе, выполняемом непосредственно на машине сервера (возможно, с помощью клиента telnet). Однако из-за того что сценарии CGI выполняются как программы, для них существуют некоторые особые требования к установке, отличные от обычных файлов HTML. В частности, их обычно требуется хранить и именовать специальным образом, и они должны быть настроены как программы, выполнять которые разрешено любым пользователям. В зависимости от потребностей после загрузки может потребоваться содействие сценариям CGI в поиске импортируемых модулей, а также преобразование их в формат текстовых файлов, соответствующий платформе сервера. Рассмотрим каждое ограничение установки более подробно:

Соглашения по каталогам и именам файлов

Прежде всего, сценарии CGI должны быть помещены в каталог, который ваш веб-сервер признает как каталог программ, и им должны даваться имена, которые ваш сервер распознает как сценарии CGI. На сервере, где расположены эти примеры, сценарии CGI могут храниться в каталоге *public_html* каждого пользователя так же, как файлы HTML, но должны иметь имена файлов, оканчивающиеся суффиксом *.cgi*, а не *.py*. Некоторые серверы допускают и суффиксы имен файлов *.py* и могут распознавать другие каталоги программ (обычно *cgi-bin*), но в этом также могут быть сильные различия и иногда допускаются настройки в зависимости от сервера и пользователя.

Соглашения по выполнению

Поскольку сценарии CGI должны выполняться веб-сервером от имени произвольных пользователей Сети, их файлам также требуется дать права доступа исполняемых файлов, чтобы пометить их как программы, и разрешение на выполнение должно быть дано остальным пользователям. На большинстве серверов это можно сделать командой оболочки `chmod 0755 имя_файла`. Сценариям CGI также обычно тре-

буется специальная строка #! в начале, указывающая на интерпретатор Python для выполнения кода файла. Текст после #! в первой строке просто указывает путь каталога к исполняемому файлу Python на машине сервера. Подробнее об этой специальной первой строке читайте в главе 2 «Системные инструменты», а если ваш сервер работает не под Unix, проверьте, какие соглашения существуют на нем.

Следует отметить одну тонкость. Как было показано ранее, специальная первая строка в выполняемых текстовых файлах обычно может содержать либо жестко заданный путь к интерпретатору Python (например, *#!/usr/bin/python*), либо вызов программы env (например, *#!/usr/bin/env python*), которая определяет местонахождение Python по значениям переменных окружения (например, $PATH). Однако прием с env менее полезен в сценариях CGI, так как значения их переменных окружения будут соответствовать пользователю «nobody» (а не вашим настройкам), что разъясняется в следующем абзаце.

Настройка пути поиска модулей (необязательная)

HTTP-серверы по соображениям безопасности обычно выполняют CGI-сценарии с именем пользователя «nobody» (это ограничивает доступ пользователя к машине сервера). Поэтому для файлов, публикуемых в Сети, должны быть установлены специальные права, делающие их доступными другим пользователям. Это также означает, что сценарии CGI не могут полагаться на то, что путь поиска модулей Python будет установлен каким-либо особым образом. Как мы видели, путь для модулей обычно инициализируется согласно значению PYTHONPATH для пользователя и значениям по умолчанию. Но так как сценарии CGI выполняются пользователем «nobody», PYTHONPATH может иметь произвольное значение при выполнении CGI-сценария.

Не ломайте себе над этим голову, поскольку часто с этим не возникает проблемы на практике. Поскольку по умолчанию Python обычно ищет импортируемые модули в текущем каталоге, проблем вообще не будет, если все сценарии и любые используемые ими модули и пакеты хранятся в вашем веб-каталоге (такая структура существует на сайте книги). Но если модуль находится в другом месте, может потребоваться редактирование списка sys.path в сценариях, чтобы вручную настроить путь поиска перед импортом (например, с помощью вызовов sys.path.append(*dirname*), присваиваний по индексам и т. д.).

Соглашения по маркерам конца строки (необязательные)

Наконец, на некоторых серверах Unix (и Linux) может также потребоваться обеспечить соответствие текстовых файлов сценариев соглашениям Unix по концу строки (\n), а не DOS (\r\n). Проблем не возникает, если редактирование и отладку производить прямо на сервере (или другой машине Unix) или один за другим передавать файлы по FTP в текстовом режиме. Но если редактировать сценарии на PC и загружать файлы на сервер Unix в tar-архиве (или в двоичном режиме FTP), то после загрузки может потребоваться преобразовать символы конца строки. Например, сервер, который использовался для разработки этого текста, возвращает установленную по умолчанию страницу сообщения об ошибке для сценариев, в которых конец строки имеет формат DOS (см. ниже сценарий конвертера).

На первый взгляд, этот процесс установки может показаться несколько сложным, но когда вы разберетесь с ним самостоятельно, все окажется не так страшно: проблемы могут возникнуть только при установке и обычно в некоторой мере могут быть автоматизированы с помощью сценариев Python, выполняемых на сервере.

Подведем итоги. Большинство сценариев Python CGI является текстовыми файлами, содержащими код Python, которые:

- именуются согласно правилам веб-сервера (например, *file.cgi*)
- хранятся в каталоге, распознаваемом веб-сервером (например, *cgi-bin/*)
- имеют права доступа исполняемых файлов (например, `chmod 755 file.cgi`)
- обычно содержат в начале особую строку `#!pythonpath` (но не `env`)
- настраивают `sys.path`, только чтобы увидеть модули в других каталогах
- используют соглашения Unix по концу строки, только если сервер не принимает формат DOS
- выводят заголовки и HTML для генерирования страницы ответа, если она требуется
- используют модуль `cgi` для анализа входных данных форм, если они есть (о формах – ниже в этой главе)

Даже при необходимости использовать машину сервера, которую настраивает кто-то другой, установление большинства используемых на машине соглашений не должно составить сложности. Например, на некоторых серверах можно переименовать этот пример в *test0.py*, и он по-прежнему будет выполняться при обращении к нему. На других серверах при обращении к файлу вам может быть выведен его исходный текст во всплывающем окне текстового редактора. Попробуйте использовать суффикс *.cgi*, если текст не выполняется, а отображается. Соглашения по каталогу CGI тоже могут различаться, но сначала попробуйте каталог, где вы обычно храните файлы HTML. И всегда нужно принимать во внимание соглашения, действующие на любой машине, куда вы планируете скопировать эти файлы примеров.

Автоматизация установки

Но погодите, зачем создавать себе трудности? Прежде чем устанавливать сценарии вручную, вспомните, что программы Python обычно могут выполнить за вас большую часть работы. Нетрудно написать сценарии Python, которые автоматизируют некоторые этапы установки CGI с помощью инструментов операционных систем, с которыми мы уже познакомились в этой книге.

Например, во время разработки примеров этой главы я осуществлял все редактирование на своем PC (обычно это более надежно, чем работать с клиентом telnet). Для инсталляции я помещаю все примеры в файл `tar`, который загружается на сервер Linux за один шаг с помощью FTP. К сожалению, мой сервер предполагает, что у сценариев CGI маркеры конца строки соответствуют Unix, а не DOS, а при распаковке файла `tar` не выполняется преобразование символов конца строки и не сохраняется разрешение на выполнение. Но вместо того чтобы выслеживать все сценарии CGI и исправлять их вручную, я просто запускаю сценарий Python примера 12.3 из команды Unix *find* после каждой загрузки на сервер.

Пример 12.3. P2E\Internet\Cgi-Wb\fixcgi.py

```
#########################################################################################
# Запуск из команды unix find для автоматизации некоторых этапов установки сценанариев cgi;
# пример:  find . -name "*.cgi" -print -exec python fixcgi.py \{} \;
# -- преобразует все сценарии cgi в формат unix перевода строки (требуется в starship)
# и задает для всех файлов cgi режим исполнения, иначе не запустятся ;
# также выполните: chmod 777 PyErrata/DbaseFiles/*, vi Extern/Email/mailconfig*;
# родственные: fixsitename.py, PyTools/fixeoln*.py, System/Filetools
#########################################################################################

# после: ungzip, untar, cp -r Cgi-Web/* ~/public_html

import sys, string, os
```

```
fname = sys.argv[1]
old   = open(fname, 'rb').read()
new   = string.replace(old, '\r\n', '\n')
open(fname, 'wb').write(new)
if fname[-3:] == 'cgi': os.chmod(fname, 0755)       # обратите внимание на восьмеричные целые:
rwx,sgo
```

Этот сценарий запускается в вершине каталога *Cgi-Web* с помощью команды оболочки Unix `csh`, которая должна применить его ко всем файлам CGI в дереве каталогов, например:

```
% find . -name "*.cgi" -print -exec python fixcgi.py \{} \;
./Basics/languages-src.cgi
./Basics/getfile.cgi
./Basics/languages.cgi
./Basics/languages2.cgi
./Basics/languages2reply.cgi
./Basics/putfile.cgi
...еще строки...
```

Вспомните из главы 2, что есть много способов обхода деревьев каталогов и поиска соответствующих файлов с помощью чистого кода Python, в том числе модуля `find`, `os.path.walk` и еще одного, рассматриваемого в сценарии из следующего раздела. Например, более переносимый альтернативный код чистого Python можно запустить так:

```
C:\...\PP2E\Internet\Cgi-Web>python
>>> import os
>>> from PP2E.PyTools.find import find
>>> for filename in find('*.cgi', '.'):
...     print filename
...     stat = os.system('python fixcgi.py ' + filename)
...
.\Basics\getfile.cgi
.\Basics\languages-src.cgi
.\Basics\languages.cgi
.\Basics\languages2.cgi
...еще строки...
```

Команда Unix *find* делает то же самое, но за пределами области видимости Python: командная строка после `-exec` выполняется для каждого найденного файла. Дополнительные подробности о команде `find` смотрите на ее странице руководства. В сценарии Python `string.replace` транслирует в маркеры конца строки Unix, а `os.chmod` действует как команда оболочки *chmod*. Существуют и другие способы трансляции маркеров конца строки; см. главу 5 «Более крупные системные примеры, часть 2».

Автоматизация редактирования при перемещении сайта

Если говорить о проблемах установки, то часто встречающимся подводным камнем веб-программирования является неработоспособность встроенных в код HTML и жестко закодированных ссылок на сайты при перемещении сайта на новый сервер. Минимальные URL (состоящие из одного имени файла) переносятся лучше, но по разным причинам используются не всегда. В один прекрасный день мне также надоело обновлять URL в гиперссылках и действиях форм, и я написал сценарий Python, который делает это вместо меня (см. пример 12.4).

Пример 12.4. PP2E\Internet\Cgi-Web\fixsitename.py

```
#!/usr/bin/env python
```

```
##############################################################################################
# Выполните этот сценарий в каталоге Cgi-Web после копирования web-примеров книги на новый
# сервер – все обращения к серверу starship в гиперссылках и тегах форм action будут
# автоматически изменены на новый сервер/сайт; предупреждает о ссылках, которые не были
# изменены (может потребоваться редактирование вручную); обратите внимание, что ссылки
# на starship обычно не нужны и не используются – т. к. броузеры помнят адреса, сервер
# и путь обычно можно опустить в URL, если предыдущая страница находится в том же месте
# (например, предполагается, что для "file.cgi" сервер/путь такие же, как для страницы,
# содержащей это имя, и настоящий URL типа "http://предыдущийсервер/предыдущийпуть/file.cgi"),
# но часть URL указана в примерах полностью; повторно использует класс Visitor, разработанный
# в главах системных средств, для обхода и преобразования всех файлов в текущем каталоге и ниже;
##############################################################################################

import os, string
from PP2E.PyTools.visitor import FileVisitor   # оболочка os.path.walk

listonly = 0
oldsite  = 'starship.python.net/~lutz'         # сервер/корневой каталог книги
newsite  = 'XXXXXX/YYYYYY'                     # замените своим сайтом
warnof   = ['starship.python', 'lutz']         # предупредить, если останется после исправления
fixext   = ['.py', '.html', '.cgi']            # типы проверяемых файлов

class FixStarship(FileVisitor):
    def __init__(self, listonly=0):                        # заменить ссылки на старые сайты
        FileVisitor.__init__(self, listonly=listonly)      # во всех текстовых файлах веб
        self.changed, self.warning = [], []                # здесь нужны разные списки
    def visitfile(self, fname):                            # или список find.find
        FileVisitor.visitfile(self, fname)
        if self.listonly:
            return
        if os.path.splitext(fname)[1] in fixext:
            text = open(fname, 'r').read()
            if string.find(text, oldsite) != -1:
                text = string.replace(text, oldsite, newsite)
                open(fname, 'w').write(text)
                self.changed.append(fname)
            for word in warnof:
                if string.find(text, word) != -1:
                    self.warning.append(fname); break

if __name__ == '__main__':
    # при щелчке не выполнять автономно
    go = raw_input('This script changes site in all web files; continue?')
    if go != 'y':
        raw_input('Canceled - hit enter key')
    else:
        walker = FixStarship(listonly)
        walker.run()
        print 'Visited %d files and %d dirs' % (walker.fcount, walker.dcount)

        def showhistory(label, flist):
            print '\n%s in %d files:' % (label, len(flist))
            for fname in flist:
                print '=>', fname
        showhistory('Made changes', walker.changed)
        showhistory('Saw warnings', walker.warning)

        def edithistory(flist):
            for fname in flist:                            # ваш редактор
                os.system('vi ' + fname)
```

```
    if raw_input('Edit changes?') == 'y':  edithistory(walker.changed)
    if raw_input('Edit warnings?') == 'y': edithistory(walker.warning)
```

Этот более сложный сценарий повторно использует модуль *visitor.py*, который мы написали в главе 5 как оболочку вызова `os.path.walk`. Если вы читали эту главу, то данный сценарий должен быть понятен. Если нет, то мы не станем здесь снова особенно вдаваться в детали. Скажем лишь, что эта программа обходит все файлы с исходным кодом в каталоге, где она выполняется, и ниже него, глобально заменяя все встретившиеся *starship.python.net/~lutz* значением, которое присвоено в сценарии переменной `newsite`. По запросу она также откроет в вашем редакторе измененные файлы и файлы, содержащие потенциально опасные строки. В данном случае в конце программы запускается текстовый редактор Unix vi, но вы можете заменить его любым другим редактором (в конце концов, это Python):

```
C:\...\PP2E\Internet\Cgi-Web>python fixsitename.py
This script changes site in all web files; continue?y
. ...
1 => .\PyInternetDemos.html
2 => .\README.txt
3 => .\fixcgi.py
4 => .\fixsitename.py
5 => .\index.html
6 => .\python_snake_ora.gif
.\Basics ...
7 => .\Basics\mlutz.jpg
8 => .\Basics\languages.html
9 => .\Basics\languages-src.cgi
...еще строки...
146 => .\PyMailCgi\temp\secret.doc.txt
Visited 146 files and 16 dirs

Made changes in 8 files:
=> .\fixsitename.py
=> .\Basics\languages.cgi
=> .\Basics\test3.html
=> .\Basics\test0.py
=> .\Basics\test0.cgi
=> .\Basics\test5c.html
=> .\PyMailCgi\commonhtml.py
=> .\PyMailCgi\sendurl.py

Saw warnings in 14 files:
=> .\PyInternetDemos.html
=> .\fixsitename.py
=> .\index.html
=> .\Basics\languages.cgi
...еще строки...
=> .\PyMailCgi\pymailcgi.html
=> .\PyMailCgi\commonhtml.py
=> .\PyMailCgi\sendurl.py
Edit changes?n
Edit warnings?y
```

В суммарном итоге этот сценарий автоматизирует часть задачи перемещения сайта: при его выполнении URL всех страниц автоматически заменяются новым именем сайта, что раздражает значительно меньше, чем поиск и редактирование всех таких ссылок вручную.

Жестко прошитых ссылок на сайт *starship* в веб-примерах этой книги немного (выше показано, что сценарий нашел и исправил восемь таких ссылок), но после копирования примеров книги на собственный сайт обязательно выполните этот сценарий из командной строки в каталоге *Cgi-Web*. Для перемещения с помощью этого сценария других сайтов просто установите надлежащим образом обе переменные — `oldsite` и `newsite`. Честолюбивый сценарист может даже запускать такой сценарий из другого, который сначала копирует содержимое сайта с помощью FTP (см. `ftplib` в предыдущей главе).[1]

Как найти Python на сервере

Последняя подсказка по установке: в контексте веб-приложения на стороне сервера не требуется установка Python у *клиентов*, но присутствовать на машине *сервера*, где должны выполняться ваши сценарии CGI, он должен. Если вы пользуетесь веб-сервером, который настраивали не сами, необходимо убедиться, что Python на этой машине есть. Более того, необходимо узнать, где он находится на этой машине, чтобы задать путь к нему в строке `#!` в начале своего сценария.

Сейчас Python является широко распространенным инструментом, поэтому обычно это не такая большая проблема, какой она была ранее. С течением времени Python еще более часто будет встречаться как стандартное средство на машинах серверов. Но если вы не уверены, есть ли на вашем сервере Python и где он находится, вот несколько советов:

- В особенности на системах Unix, следует сначала предположить, что Python находится в стандартном месте (например, */usr/local/bin/python*), и проверить его работоспособность. Весьма вероятно, что Python на этих машинах уже есть. Если у вас есть telnet-доступ к своему серверу, можно воспользоваться командой Unix *find*, начав с каталога */usr*.
- Если ваш сервер работает под Linux, то, вероятно, все готово к работе. В настоящее время Python стандартно входит в дистрибутивы Linux, а под операционной системой Linux работают многие сайты и провайдеры услуг Интернета; на таких сайтах Python, вероятно, уже установлен в */usr/bin/python*.
- В других средах, где нет возможности самостоятельно контролировать машину сервера, получить доступ к уже установленному Python может оказаться труднее. В этих случаях можно переместить свой сайт на сервер, где Python точно установлен, уговорить провайдера установить Python на машине, которую вы хотите использовать, или установить Python на машине сервера самому.

Если ваш провайдер без симпатии относится к вашей потребности в Python и вы хотите перенести свой сайт к тому провайдеру, который предоставляет возможность его использования, можно найти списки дружественно относящихся к Python провайдеров на *http://www.python.org*. А если вы предпочтете установить Python на машине сервера самостоятельно, возьмите средство *freeze* («заморозка»), поставляемое с дистрибутивом исходного кода Python (находится в каталоге *Tools*). С помощью *freeze* можно создать единственный файл выполняемой программы, который целиком содержит интерпретатор Python вместе со всеми стандартными библиотечными модулями. Такой «за-

[1] Как отмечалось в начале главы, часто есть несколько способов решения задач веб-мастеринга. Например, тег HTML `<BASE>` может дать другой способ преобразования абсолютных URL, а передача файлов сайта на сервер по FTP каждого в отдельности и в текстовом режиме поможет избежать проблем с символами конца строки. Несомненно, существуют и другие способы решения таких задач. Хотя такие альтернативы принесли бы мало пользы книге, иллюстрирующей приемы программирования на Python.

мороженный» интерпретатор можно за один шаг загрузить на вашу учетную запись на сервере по FTP, при этом не потребуется полной инсталляции Python на сервере.

Добавление картинок и генерация таблиц

Вернемся снова к написанию кода для сервера. Каждому, когда-либо бродившему по Сети, известно, что веб-страницы обычно содержат не только обычный текст. В примере 12.5 представлен CGI-сценарий Python, который помещает в вывод тег HTML `<IMG>` для создания графического образа в броузере клиента. Чего-либо специфического для Python в этом примере нет, но обратите внимание, что графический файл (*ppsmall.gif*) располагается на сервере и загружается с него при интерпретировании броузером вывода этого сценария так же, как простые файлы HTML.

Пример 12.5. PP2E\Internet\Cgi-Web\Basics\test1.cgi

```
#!/usr/bin/python

text = """Content-type: text/html

<TITLE>CGI 101</TITLE>
<H1>A Second CGI script</H1>
<HR>
<P>Hello, CGI World!</P>
<IMG src="ppsmall.gif" BORDER=1 ALT=[image]>
<HR>
"""

print text
```

Обратите внимание на использование здесь блока строк в тройных кавычках; вся строка HTML посылается броузеру в один прием с помощью находящегося в конце оператора print. Если функционируют и клиент и сервер, при ссылке на этот сценарий и его выполнении будет сгенерирована страница, выглядящая, как на рис. 12.4.

До сих пор наши сценарии CGI выводили готовый HTML, который с таким же успехом можно было бы хранить в файле HTML. Но поскольку CGI-сценарии являются

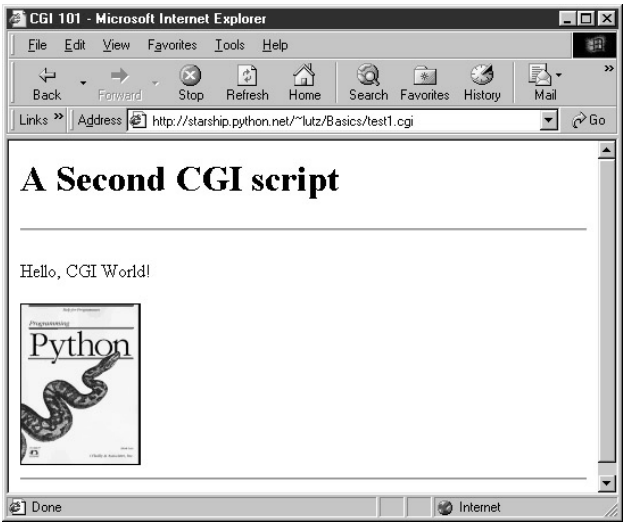


Рис. 12.4. Страница с изображением, генерируемая test1.cgi

выполняемыми программами, с их помощью можно генерировать HTML на лету, динамически – даже, возможно, как ответ на определенный набор введенных пользователем данных, переданных сценарию. В конце концов, в этом и состоит назначение сценариев CGI. Давайте теперь и воспользуемся этим и напишем сценарий Python, который строит HTML ответа программным образом (см. пример 12.6).

Пример 12.6. PP2E\Internet\Cgi-Web\Basics\test2.cgi

```
#!/usr/bin/python

print """Content-type: text/html

<TITLE>CGI 101</TITLE>
<H1>A Third CGI script</H1>
<HR>
<P>Hello, CGI World!</P>

<table border=1>
"""

for i in range(5):
    print "<tr>"
    for j in range(4):
        print "<td>%d.%d</td>" % (i, j)
    print "</tr>"

print """
</table>
<HR>
"""
```

Несмотря на все теги, это действительно код Python – сценарий *test2.cgi* снова встраивает блоки HTML с помощью строк, заключенных в тройные кавычки. На этот раз вложенные циклы Python `for` динамически генерируют часть HTML, посылаемого броузеру. Конкретно, выдается HTML, размещающий в середине страницы двумерную таблицу, как на рис. 12.5.

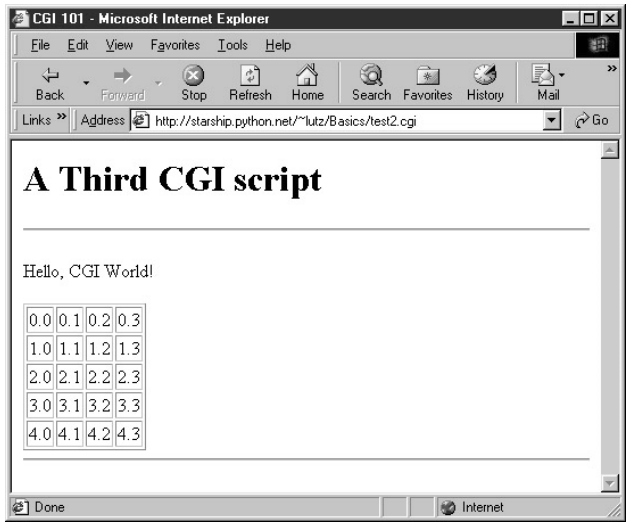


Рис. 12.5. Страница с таблицей, генерируемая test2.cgi

В каждой строке таблицы выводится пара «строка.колонка», сгенерированная при выполнении сценария Python. Если вам интересно узнать, как выглядит сгенерированный HTML, откройте страницу в броузере и выберите в меню пункт View Source. Это одна страница HTML, сгенерированного первым `print` сценария, затем циклами `for` и наконец последним `print`. Иными словами, объединение вывода этого сценария представляет собой HTML-документ с заголовками.

Теги таблиц

Этот сценарий генерирует теги таблиц HTML. Опять-таки, мы не собираемся здесь изучать HTML, но бегло рассмотрим его, чтобы вы могли понять этот пример. Таблицы в HTML объявляются как текст между тегами `<table>` и `</table>`. Обычно текст таблицы, в свою очередь, объявляет содержимое каждой строки таблицы между тегами `<tr>` и `</tr>` и каждой колонки между тегами `<td>` и `</td>`. Циклы в нашем сценарии строят HTML, объявляющий пять строк по четыре колонки в каждой, путем вывода соответствующих тегов, при этом значениями ячеек являются номера текущих строки и колонки. Например, вот часть вывода сценария, определяющая первые две строки:

```
<table border=1>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
</tr>
. . .
</table>
```

Другие теги и параметры таблиц позволяют задавать строки заголовков (`<th>`), тип границ и т. д. В одном из следующих разделов мы еще рассмотрим применение синтаксиса таблиц для структурирования форм.

Добавление взаимодействия с пользователем

Сценарии CGI отлично умеют генерировать HTML на лету подобным образом, но их также часто используют для осуществления взаимодействия с пользователем, вводящим данные в веб-броузере. Как отмечалось ранее в этой главе, взаимодействие с веб обычно осуществляется с помощью двухэтапного процесса и двух разных веб-страниц: заполняется страница и нажимается кнопка submit, а в ответ возвращается новая страница. В промежутке данные формы обрабатывает сценарий CGI.

Передача данных

Это описание выглядит достаточно просто, но процедура получения данных, введенных пользователем, требует понимания специального тега HTML, `<form>`. Обратимся к реализации простого взаимодействия в веб, чтобы посмотреть, как действуют формы. Сначала нужно создать описание страницы формы, заполняемой пользователем, приведенное в примере 12.7.

Пример 12.7. PP2E\Internet\Cgi-Web\Basics\test3.html

```
<html><body>
<title>CGI 101</title>
<H1>A first user interaction: forms</H1>
<hr>
<form method=POST action="http://starship.python.net/~lutz/Basics/test3.cgi">
    <P><B>Enter your name:</B>
    <P><input type=text name=user>
    <P><input type=submit>
</form>
</BODY></HTML>
```

test3.html является простым файлом HTML, а не сценарием CGI (хотя его содержимое можно было бы вывести и в сценарии). При обращении к этому файлу текст между тегами `<form>` и `</form>` генерирует поля ввода и кнопку Submit, показанные на рис. 12.6.

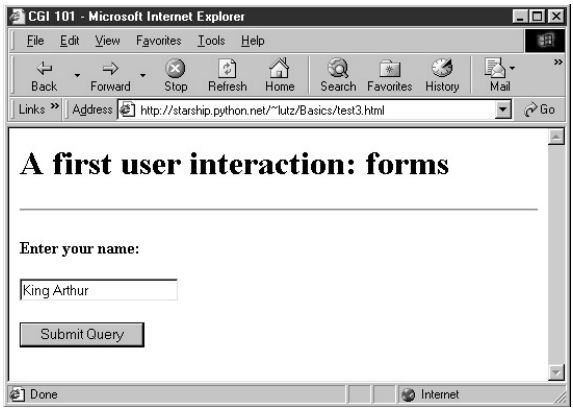


Рис. 12.6. Страница простой формы, генерируемая test3.html

Дополнительно о тегах формы

Мы не станем подробно обсуждать кодирование форм HTML, но подчеркнем некоторые места. Внутри кода HTML-формы:

- Параметр формы `action` задает URL CGI-сценария, который будет вызван для обработки отправленных данных формы. Это ссылка формы на обрабатывающую ее программу – в данном случае программу *test3.cgi* в моем исходном каталоге веб на машине сервера с именем *starship.python.net*. Параметр `action` близок по духу параметру `command` в кнопках Tkinter – это место, где для броузера регистрируется обработчик обратного вызова (в данном случае удаленный обработчик).
- Управляющие элементы для ввода задаются вложенными тегами `<input>`. В данном примере у тегов ввода есть два главных параметра. Параметр *type* принимает значения `text` для текстовых полей и `submit` для кнопки Submit (которая отправляет данные серверу и по умолчанию имеет метку «Submit Query»). Параметр *name* служит для идентификации введенного значения по ключу, когда данные формы будут получены сервером. Например, серверный сценарий CGI, который мы увидим чуть ниже, использует строку `user` в качестве ключа для получения данных, введенных в текстовое поле этой формы. Как будет показано в последующих примерах, другие параметры тега `input` могут задавать начальные значения (`value=X`), режим «только

для чтения» (`readonly`) и т. д. Другие значения параметра *type* могут передавать скрытые данные (`type=hidden`), повторно инициализировать поля (`type=reset`) или создавать кнопки для выбора нескольких вариантов (`type=checkbox`).

- У форм есть также параметр `method`, задающий стиль кодировки, используемой при передаче данных через сокет на целевую машину сервера. В данном случае используется стиль `post`, при котором устанавливается связь с сервером, а затем ему в отдельной передаче посылается поток данных, введенных пользователем. Другим возможным вариантом является стиль `get`, при котором входная информация передается серверу за один шаг путем добавления данных пользователя в конец URL, вызывающего сценарий, обычно после символа `?` (подробнее об этом ниже). При передаче методом `get` входные данные обычно оказываются на сервере в виде переменных окружения или как аргументы в командной строке, используемой для запуска сценария. При передаче методом `post` данные считываются со стандартного ввода и раскодируются. К счастью, модуль Python `cgi` прозрачным образом обрабатывает оба стиля кодировки, поэтому нашим сценариям CGI не приходится беспокоиться о том, который из стилей использован.

Обратите внимание, что для иллюстрации URL действия в форме этого примера записан в виде полного адреса. Так как броузер помнит, откуда пришла содержащая URL страница HTML, он будет действовать так же, если указать лишь имя файла сценария, как показано в примере 12.8.

Пример 12.8. PP2E\Internet\Cgi-Web\Basics\test3-minimal.html

```
<html><body>
<title>CGI 101</title>
<H1>A first user interaction: forms</H1>
<hr>
<form method=POST action="test3.cgi">
    <P><B>Enter your name:</B>
    <P><input type=text name=user>
    <P><input type=submit>
</form>
</BODY></HTML>
```

Полезно помнить, что URL, встраиваемые в теги форм action и гиперссылки, служат адресами в первую очередь для броузера, а не сценария. Сам сценарий *test3.cgi* безразличен к тому, какого вида URL его запустил – минимальный или полный. На самом деле все части URL, включая имя файла сценария (и вплоть до параметров запроса URL) участвуют в диалоге между броузером и HTTP-сервером до того, как будет запущен сценарий CGI. Если броузер знает, с каким сервером нужно связаться, то URL сработает, но URL, поступающие из-за пределов страницы (например, введенные в поле адреса броузера или отправляемые модулю Python `urllib`), обычно должны быть заданы полностью, потому что понятия предыдущей страницы не существует.

Ответ

Пока мы создали только одну статическую страницу с полем ввода. Но кнопка Submit на этой странице творит чудеса. При нажатии на нее запускается удаленная программа, URL которой указан в параметре формы `action`, и этой программе передаются данные, введенные пользователем, в соответствии с параметром формы `method`, определяющим стиль кодировки. Пока пользователь на клиентской машине ждет ответа, на сервере запускается сценарий Python, обрабатывающий данные, введенные в форму, и показанный в примере 12.9.

Пример 12.9. PP2E\Internet\Cgi-Web\Basics\test3.cgi

```
#!/usr/bin/python
########################################################################
# Выполняется на сервере, читает данные, введенные в форму, выводит html;
# url=http://server-name/root-dir/Basics/test3.cgi
########################################################################

import cgi
form = cgi.FieldStorage()            # анализ данных формы
print "Content-type: text/html"      # плюс пустая строка

html = """
<TITLE>test3.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<P>%s</P>
<HR>"""

if not form.has_key('user'):
    print html % "Who are you?"
else:
    print html % ("Hello, %s." % form['user'].value)
```

Как и ранее, этот CGI-сценарий Python выводит HTML, генерирующий страницу ответа в броузере клиента. Но этот сценарий делает еще кое-что: он использует стандартный модуль `cgi` для анализа данных, введенных пользователем на предыдущей веб-странице (см. рис. 12.6). К счастью, в Python это происходит автоматически: обращение к классу `FieldStorage` модуля `cgi` автоматически выполняет всю работу по извлечению данных формы из входного потока и переменных окружения независимо от способа передачи этих данных – в потоке в стиле `post` или добавляемых к URL параметрах в стиле `get`. Вводимые данные, пересылаемые в любом из стилей, выглядят для сценариев Python одинаково.

Сценарии должны один раз вызвать `cgi.FieldStoreage` перед тем, как обращаться к значениям полей. В результате этого вызова возвращается объект, имеющий вид словаря – поля для ввода данных пользователем из формы (или URL) представляются в виде значений ключей этого объекта. Например, `form['user']` в сценарии является объектом, атрибут `value` которого представляет собой строку, содержащую текст, введенный в текстовое поле формы. Если вы перелистаете книгу назад к HTML-коду страницы формы, то заметите, что параметр `name` поля ввода имел значение `user` – имя в HTML формы стало ключом, по которому введенное значение извлекается из словаря. Объект, возвращаемый `FieldStorage`, поддерживает и другие операции со словарем, например с помощью метода `has_key` можно проверить, есть ли некоторое поле во входных данных.

Перед своим завершением этот сценарий выводит HTML, создающий страницу результата, на которой повторяются данные, введенные пользователем в форму. Два выражения форматирования строк (`%`) применяются для вставки введенного текста в строку ответа, а строки ответа – в заключенный в тройные кавычки блок строк HTML. Тело вывода сценария выглядит так:

```
<TITLE>test3.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<P>Hello, King Arthur.</P>
<HR>
```

В броузере этот вывод превращается в страницу, показанную на рис. 12.7.

Рис. 12.7. Результат обработки параметров формы сценарием test3.cgi

Передача параметров в URL

Обратите внимание, что вверху броузера показывается URL сценария, сгенерировавшего эту страницу. Сам этот URL мы не вводили – он появился из атрибута `action` тега `form` в HTML предыдущей страницы. Однако ничто не мешает нам ввести URL сценария явным образом в поле адреса броузера, чтобы запустить сценарий, как мы это делали ранее для примеров сценария CGI и файла HTML.

Но здесь есть ловушка: откуда возьмется значение поля ввода, если страницы с формой нет? То есть, если ввести URL сценария CGI самостоятельно, как будет заполнено поле ввода? Ранее, говоря о форматах URL, я сказал, что в схеме кодировки `get` входные параметры помещаются в конец URL. При явном вводе адресов сценариев тоже можно добавлять входные значения в конец URL, где они служат той же цели, что и поля `<input>` в формах. Кроме того, модуль Python `cgi` обеспечивает одинаковое представление в сценариях данных, полученных из URL и форм.

Например, можно вообще пропустить заполнение страницы с формой и прямо вызвать сценарий *test3.cgi*, обратившись к URL вида:

```
http://starship.python.net/~lutz/Basics/test3.cgi?user=Brian
```

В этом URL значение поля ввода с именем `user` задано явно, как если бы пользователь заполнил страницу ввода. При вызове подобным образом единственным ограничением является соответствие имени параметра `user` имени, ожидаемому сценарием (и жестко закодированному в HTML-коде формы). Мы используем здесь лишь один параметр, но в целом параметры URL обычно указываются как символ `?` с последующими присваиваниями `name=value`, разделяемыми символами `&`, если их больше одного. На рис. 12.8 показана страница ответа, получаемая после ввода URL с явно указанными данными.

В целом, любой сценарий CGI можно вызвать, как заполнив и передав страницу формы, так и передав входные данные в конце URL. Трудно не обратить внимание на сходство между вызовом подобным образом сценариев CGI с явными входными параметрами и *функциями*, хотя и располагающимися удаленно в сети. Передача данных сценариям через URL аналогична аргументам ключевых слов в функциях Python, как по действию, так и синтаксически. На самом деле в главе 15 мы познакомимся с системой под названием Zope, которая делает связь между URL и вызовами функций в Python еще более точной (URL становятся более непосредственными вызовами функций).

Между прочим, если очистить поле ввода с именем на форме ввода (то есть сделать его пустым) и нажать кнопку передачи, то поле имени `user` окажется пустым. Точнее, броузер может вообще не послать это поле вместе с другими данными формы, несмот-

Рис. 12.8. Результат обработки параметров URL сценарием test3.cgi

Рис. 12.9. Пустое поле имени порождает страницу сообщения об ошибке

ря на то, что оно перечислено в HTML, описывающем структуру формы. Сценарий CGI обнаруживает такое отсутствующее поле с помощью метода словаря `has_key` и порождает в ответ страницу, показанную на рис. 12.9.

Вообще, сценарии CGI должны проверять отсутствие каких-либо полей, поскольку пользователь мог не ввести их в форму или формы вообще нет – поля ввода не вставлены в конец явно введенного URL. Например, если ввести URL сценария вообще без параметров (то есть отбросить текст начиная с ?), будет получена та же самая страница ответа с сообщением об ошибке. Так как любой сценарий CGI может быть вызван через форму или URL, сценарии должны быть готовы к обеим ситуациям.

Задание структуры формы с помощью таблицы

Теперь возьмем что-нибудь более практическое. В большинстве приложений CGI на страницах ввода есть несколько полей. Когда полей больше одного, метки и поля для ввода обычно располагаются в виде таблицы, чтобы придать форме структурированный внешний вид. Файл HTML примера 12.10 определяет форму с двумя полями ввода.

Пример 12.10. PP2E\Internet\Cgi-Web\Basics\test4.html

```
<html><body>
<title>CGI 101</title>
<H1>A second user interaction: tables</H1>
<hr>
<form method=POST action="test4.cgi">
  <table>
    <TR>
      <TH align=right>Enter your name:
```

```
      <TD><input type=text name=user>
    <TR>
      <TH align=right>Enter your age:
      <TD><input type=text name=age>
    <TR>
      <TD colspan=2 align=center>
      <input type=submit value="Send">
  </table>
</form>
</body></html>
```

Тег <TH> определяет колонку, как и <TD>, но также помечает ее как колонку заголовка, что обычно означает вывод ее полужирным шрифтом. Размещая поля ввода и метки в такой таблице, получаем страницу ввода, выглядящую, как на рис. 12.10. Метки и поля ввода автоматически выравниваются по вертикали в колонках подобно тому, как это делают менеджеры геометрии GUI Tkinter, с которыми мы встречались ранее в этой книге.

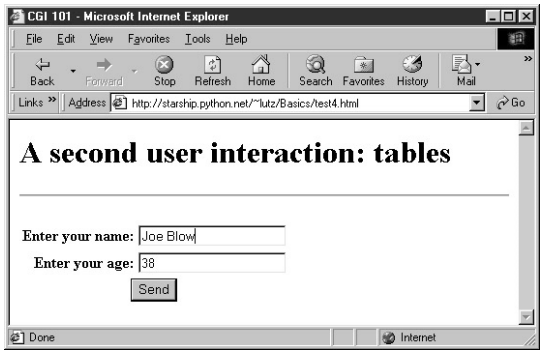


Рис. 12.10. Форма, размеченная тегами таблиц

При нажатии кнопки Submit на этой форме (помеченной как «Send» в HTML страницы) на машине сервера выполняется сценарий примера 12.11 со входными данными, введенными пользователем.

Пример 12.11. PP2E\Internet\Cgi-Web\Basics\test4.cgi

```
#!/usr/bin/python
########################################################################
# Выполняется на сервере, читает данные из формы, выводит html;
# url http://server-name/root-dir/Basics/test4.cgi
########################################################################

import cgi, sys
sys.stderr = sys.stdout              # вывод ошибок в броузер
form = cgi.FieldStorage()            # разбор данных формы
print "Content-type: text/html\n"    # плюс пустая строка

# class dummy:
#     def __init__(self, s): self.value = s
# form = {'user': dummy('bob'), 'age':dummy('10')}

html = """
<TITLE>test4.cgi</TITLE>
<H1>Greetings</H1>
<HR>
```

```
<H4>%s</H4>
<H4>%s</H4>
<H4>%s</H4>
<HR>"""

if not form.has_key('user'):
    line1 = "Who are you?"
else:
    line1 = "Hello, %s." % form['user'].value

line2 = "You're talking to a %s server." % sys.platform

line3 = ""
if form.has_key('age'):
    try:
        line3 = "Your age squared is %d!" % (int(form['age'].value) ** 2)
    except:
        line3 = "Sorry, I can't compute %s ** 2." % form['age'].value

print html % (line1, line2, line3)
```

Структура таблицы задается файлом HTML, а не этим CGI-сценарием Python. На самом деле в этом сценарии немного нового – как и прежде, с помощью форматирования строки входные значения вставляются в заключенную в тройные кавычки строку шаблона для HTML-кода страницы ответа, но на этот раз выводится отдельная строка для каждого поля ввода. Однако следует обратить внимание на несколько новых приемов, особенно в отношении отладки сценария CGI и безопасности. Мы поговорим о них в следующих двух разделах.

Преобразование строк в сценариях CGI

В порядке развлечения этот сценарий возвращает название платформы сервера, полученное от `sys.platform`, а также квадрат значения поля ввода `age`. Обратите внимание на необходимость преобразования значения `age` в целое число с помощью встроенной функции `int`; в CGI все входные значения поступают в виде строк. Преобразование в целое число можно также выполнить с помощью встроенных функций `string.atoi` или `eval`. Ошибки преобразования (и прочие) элегантно перехватываются в операторе `try`, чтобы вывести строку ошибки и не дать сценарию умереть.

Никогда нельзя преобразовывать с помощью `eval` строки, переданные через Интернет, как поле `age` в данном примере, если только нет абсолютной уверенности в том, что строка не может содержать злонамеренного кода. Например, если бы этот пример был размещен в Интернете, то кто-нибудь мог ввести в поле `age` значение вроде `os.system('rm *')` (или дописать параметр с таким значением `age` к URL). При передаче в `eval` такая строка может уничтожить все файлы в каталоге сценариев сервера!

О способах уменьшить такой риск с помощью режима Python ограниченного выполнения (модуль `rexec`) мы расскажем в главе 15. Но по умолчанию строки, получаемые из Сети, нужно рассматривать как очень опасные для сценариев CGI. Никогда не передавайте их инструментам динамического программирования, таким как `eval` или `exec`, а также средствам, выполняющим произвольные команды оболочки, например `os.popen` и `os.system`, если только вы не уверены в их безвредности, или не включили режим ограниченного выполнения Python.

Отладка сценариев CGI

Ошибки бывают, даже в прекрасном мире Интернета. Вообще говоря, отладка сценариев CGI может оказаться значительно более трудной, чем отладка программ, выполняемых локально. Ошибки происходят на удаленной машине, да и вообще сценарии не будут выполняться при отсутствии контекста, предполагаемого моделью CGI. Сценарий примера 12.11 демонстрирует следующие два распространенных приема отладки:

Перехват сообщений об ошибках

Данный сценарий назначает `sys.stderr` в `sys.stdout`, чтобы сообщения Python об ошибках в конечном счете выводились на странице ответа в броузере. Обычно сообщения Python об ошибках выводятся в `stderr`. Для того чтобы направить их в броузер, нужно заставить `stderr` ссылаться на тот же самый объект файла, что и `stdout` (который в сценариях CGI соединяется с броузером). Если не сделать такого назначения, ошибки Python, в том числе программные ошибки сценария, не появятся в броузере.

Имитация контрольного примера

Определение класса `dummy`, закомментированное в этой окончательной версии, использовалось для отладки сценария перед тем, как он был размещен в Сети. Помимо того, что сообщения `stderr` по умолчанию не видны, сценарии CGI также предполагают наличие окружающего контекста, которого нет при отладке их вне окружения CGI. Например, при выполнении из командной строки системы у этого сценария нет входных данных из формы. Раскомментируйте этот код для тестирования из командной строки системы. Класс `dummy` маскируется под объект разобранных полей формы, а `form` получает значения словаря, содержащего два объекта полей формы. В итоге `form` подменяет результат вызова `cgi.FieldStorage`. Как обычно в Python, необходимо придерживаться лишь интерфейсов объектов, а не типов данных.

Вот несколько общих рекомендаций по отладке CGI-сценариев на сервере:

Запуск сценария из командной строки

Возможно, при этом не будет сгенерирован HTML, но при автономном выполнении выявятся синтаксические ошибки в коде. Вспомните, что в командной строке Python можно запускать файлы с исходным кодом независимо от их расширений, например нормально работает команда `python somescript.cgi`.

Назначение sys.stderr в sys.stdout как можно ближе к началу своего сценария

В результате при обращении к сценарию текст сообщений об ошибках и дамп стека будут показываться в броузере клиента. В действительности, если не считать утомительный поиск в журналах сервера, это может оказаться единственным способом увидеть текст сообщения об ошибке при аварийном завершении сценария.

Имитация входных данных для моделирования окружающего контекста CGI

Например, определите классы, имитирующие интерфейс входных данных CGI (как это делает класс `dummy` в данном сценарии), чтобы посмотреть вывод сценария для разных контрольных примеров при запуске его из командной строки системы.[1] Иногда также полезно установить переменные окружения так, чтобы они

[1] Кстати, этот прием применим не только к сценариям CGI. В главе 15 мы познакомимся с системами, встраивающими Python внутрь HTML. Нет хорошего способа протестировать такой код вне контекста объемлющей системы, кроме как извлечь встроенный код Python (возможно, с помощью парсера HTML `htmllib`, поставляемого с Python) и запустить его, передав ему имитацию API, который будет использован в конечном итоге.

имитировали данные формы или URL (далее в этой главе будет показан пример такого приема).

Вызов утилит для вывода контекста CGI в броузере

В модуле CGI есть вспомогательные функции, посылающие в броузер форматированный дамп переменных окружения CGI и входных значений (например, `cgi.test`, `cgi.print_form`). Иногда этого оказывается достаточно для решения проблем соединения. Некоторые из этих функций будут использованы в конкретном примере почтовой программы в следующей главе.

Показ перехватываемых исключительных ситуаций

При перехвате возбужденной Python исключительной ситуации сообщение Python об ошибке по умолчанию не выводится в `stderr`. В таких случаях вывод названия исключительной ситуации и ее значения на странице ответа определяется вашим сценарием. Подробности, касающиеся исключительной ситуации, могут быть получены с помощью встроенного модуля `sys`. Это также будет применено в примере, приводимом ниже.

Запуск живьем

Конечно, если ваш сценарий хоть как-то работает, лучше всего запустить его живьем на сервере и передать ему реальные данные через броузер.

При запуске этого сценария в результате посылки страницы с формой для ввода он выдает данные, формирующие новую страницу ответа, показанную на рис. 12.11.

Как обычно, параметры для этого сценария CGI можно также передать в конце URL. На рис. 12.12 показана страница, получаемая при явной передаче в URL полей `user` и `age`. Обратите внимание, что на этот раз после `?` следуют два параметра, разделенные `&`. Кроме того, пробел в значении `user` задан с помощью `+`. Это стандартное соглашение по кодировке URL. На сервере `+` автоматически заменяется обратно на пробел. Это входит в стандартные правила преобразования строк URL, к которым мы вернемся позднее.

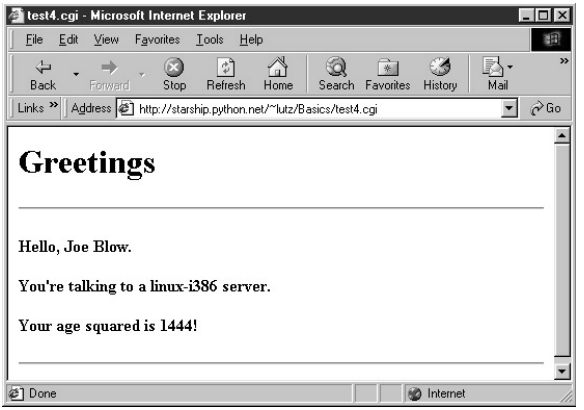


Рис. 12.11. Страница ответа, сгенерированная test4.cgi

Добавление стандартных инструментов ввода

До сих пор мы вводили данные в текстовые поля, но формы HTML поддерживают ряд элементов ввода (называемые графическими элементами – «widgets» – в традиционном мире GUI) для получения данных, вводимых пользователем. Рассмотрим про-

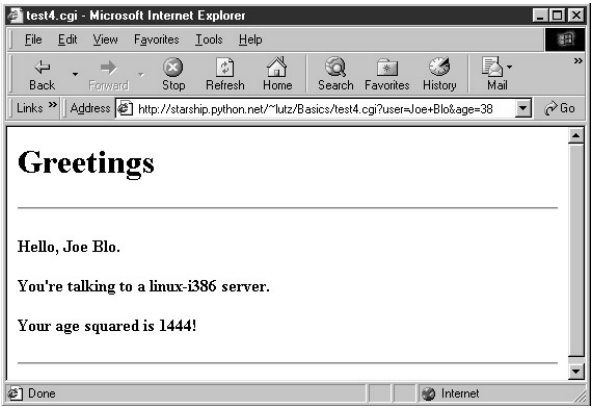


Рис. 12.12. Страница ответа, сгенерированная test4.cgi для параметров, переданных в URL

грамму CGI, которая показывает сразу все стандартные элементы для ввода. Как обычно, определим файл HTML для страницы формы и CGI-сценарий Python для обработки введенных данных и создания ответной страницы. Файл HTML представлен в примере 12.12.

Пример 12.12. PP2E\Internet\Cgi-Web\Basics\test5a.html

```
<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices</H1>
<HR>
<FORM method=POST action="test5.cgi">
  <H3>Please complete the following form and click Send</H3>
  <P><TABLE>
    <TR>
      <TH align=right>Name:
      <TD><input type=text name=name>
    <TR>
      <TH align=right>Shoe size:
      <TD><table>
      <td><input type=radio name=shoesize value=small>Small
      <td><input type=radio name=shoesize value=medium>Medium
      <td><input type=radio name=shoesize value=large>Large
      </table>
    <TR>
      <TH align=right>Occupation:
      <TD><select name=job>
        <option>Developer
        <option>Manager
        <option>Student
        <option>Evangelist
        <option>Other
      </select>
    <TR>
      <TH align=right>Political affiliations:
      <TD><table>
      <td><input type=checkbox name=language value=Python>Pythonista
      <td><input type=checkbox name=language value=Perl>Perlmonger
```

```
        <td><input type=checkbox name=language value=Tcl>Tcler
        </table>
      <TR>
        <TH align=right>Comments:
        <TD><textarea name=comment cols=30 rows=2>Enter text here</textarea>
      <TR>
        <TD colspan=2 align=center>
        <input type=submit value="Send">
    </TABLE>
  </FORM>
  <HR>
  </BODY></HTML>
```

При отображении в броузере появляется страница, показанная на рис. 12.13.

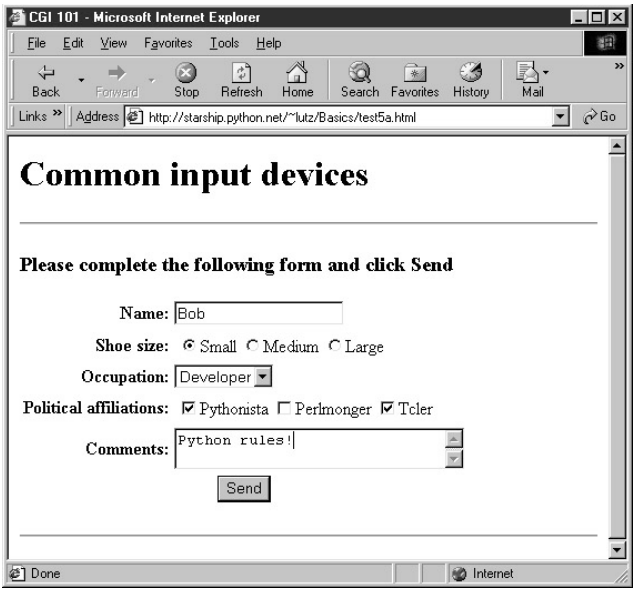


Рис. 12.13. Страница формы, генерируемая test5a.html

Как и прежде, на этой странице есть обычное текстовое поле, но, кроме того, переключатели, окно выпадающего списка, группа флажков для выбора нескольких вариантов и область для ввода многострочного текста. Для всех них в HTML-файле задан параметр `name`, идентифицирующий выбранное ими значение в данных, направляемых клиентом серверу. Если заполнить эту форму и щелкнуть по кнопке передачи Send, на сервере будет запущен сценарий из примера 12.13 для обработки всех входных данных, введенных с клавиатуры или выбранных на форме.

Пример 12.13. PP2E\Internet\Cgi-Web\Basics\test5.cgi

```
#!/usr/bin/python
##########################################################################
# Выполняется на сервере, читает введенные в форму данные, выводит html;
# url=http://server-name/root-dir/Basics/test5.cgi
##########################################################################

import cgi, sys, string
form = cgi.FieldStorage()           # анализ данных формы
```

```
print "Content-type: text/html"      # плюс пустая строка

html = """
<TITLE>test5.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<H4>Your name is %(name)s</H4>
<H4>You wear rather %(shoesize)s shoes</H4>
<H4>Your current job: %(job)s</H4>
<H4>You program in %(language)s</H4>
<H4>You also said:</H4>
<P>%(comment)s</P>
<HR>"""

data = {}
for field in ['name', 'shoesize', 'job', 'language', 'comment']:
    if not form.has_key(field):
        data[field] = '(unknown)'
    else:
        if type(form[field]) != type([]):
            data[field] = form[field].value
        else:
            values = map(lambda x: x.value, form[field])
            data[field] = string.join(values, ' and ')
print html % data
```

Этот сценарий Python выполняет немного действий; в основном он просто копирует данные полей формы в словарь с именем `data`, чтобы их можно было легко вставить в заключенную в тройные кавычки строку ответа. Стоит пояснить некоторые использованные приемы:

Проверка правильности полей

Как обычно, необходимо проверить все предполагаемые поля и убедиться в их присутствии во входных данных с помощью метода словаря `has_key`. Некоторые или все поля ввода могут отсутствовать, если на форме в них не были введены данные или они не были добавлены в явно заданный URL.

Форматирование строк

На этот раз в строке формата используются ссылки на ключи словаря – вспомните, что `%(name)s` означает необходимость извлечь значение для ключа `name` из словаря данных и выполнить для него преобразование в строку.

Поля с несколькими вариантами выбора

Проверяется также тип значений всех ожидаемых полей, чтобы определить, не получен ли вместо обычной строки список. Значения элементов ввода с несколькими вариантами выбора, таких как поле выбора `language` на этой странице ввода, возвращаются из `cgi.FieldStorage` в виде списка объектов с атрибутами `value`, а не простого одиночного объекта с `value`. Этот сценарий дословно копирует в словарь значения простых полей, но с помощью `map` получает значения полей с несколькими вариантами выбора и с помощью `string.join` строит из них одну строку, вставляя `and` между выбранными значениями (например, `Python and Tcl`).[1]

[1] Забегая вперед, сделаем два замечания. Помимо простых строк и списков мы позднее увидим третий тип объектов входных данных форм, который возвращается для полей, задающих загрузку файлов на сервер. В этом примере сценария следовало бы также для обеспечения надежности преобразовывать отражаемый эхом текст, который вставляется в HTML ответа, чтобы в нем не было операторов HTML. Подробнее преобразование будет обсуждаться ниже.

Когда эта страница с формой заполнена и передана, сценарий создает ответ, показанный на рис. 12.14, – в сущности, лишь форматированное отражение того, что было передано.

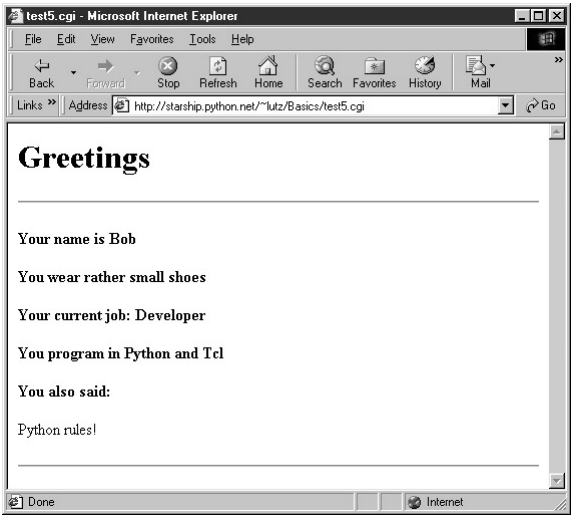


Рис. 12.14. Страница ответа, созданная test5.cgi (1)

Изменение разметки формы ввода

Предположим, что вы написали такую систему, а ваши пользователи, клиенты и «вторая половина» жалуются, что форму ввода трудно читать. Не волнуйтесь. Поскольку модель CGI естественным образом отделяет *интерфейс пользователя* (описание страницы HTML) от *обрабатывающей логики* (сценария CGI), можно совершенно безболезненно изменить структуру формы. Модифицируйте файл HTML, а изменять код CGI нет никакой необходимости. Скажем, в примере 12.14 приводится новое определение входной формы, в котором несколько иным образом используются таблицы, обеспечивая более красивую разметку с границами.

Пример 12.14. PP2E\Internet\Cgi-Web\Basics\test5b.html

```
<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices: alternative layout</H1>
<P>Use the same test5.cgi server side script, but change the
layout of the form itself.  Notice the separation of user interface
and processing logic here; the CGI script is independent of the
HTML used to interact with the user/client.</P><HR>

<FORM method=POST action="test5.cgi">
  <H3>Please complete the following form and click Submit</H3>
  <P><TABLE border cellpadding=3>
    <TR>
      <TH align=right>Name:
      <TD><input type=text name=name>
    <TR>
      <TH align=right>Shoe size:
      <TD><input type=radio name=shoesize value=small>Small
          <input type=radio name=shoesize value=medium>Medium
```

```
        <input type=radio name=shoesize value=large>Large
    <TR>
      <TH align=right>Occupation:
      <TD><select name=job>
        <option>Developer
        <option>Manager
        <option>Student
        <option>Evangelist
        <option>Other
      </select>
    <TR>
      <TH align=right>Political affiliations:
      <TD><P><input type=checkbox name=language value=Python>Pythonista
          <P><input type=checkbox name=language value=Perl>Perlmonger
          <P><input type=checkbox name=language value=Tcl>Tcler
    <TR>
      <TH align=right>Comments:
      <TD><textarea name=comment cols=30 rows=2>Enter spam here</textarea>
    <TR>
      <TD colspan=2 align=center>
      <input type=submit value="Submit">
      <input type=reset  value="Reset">
  </TABLE>
</FORM>
</BODY></HTML>
```

Пойдя в броузере на эту альтернативную страницу, мы получим интерфейс, показанный на рис. 12.15.

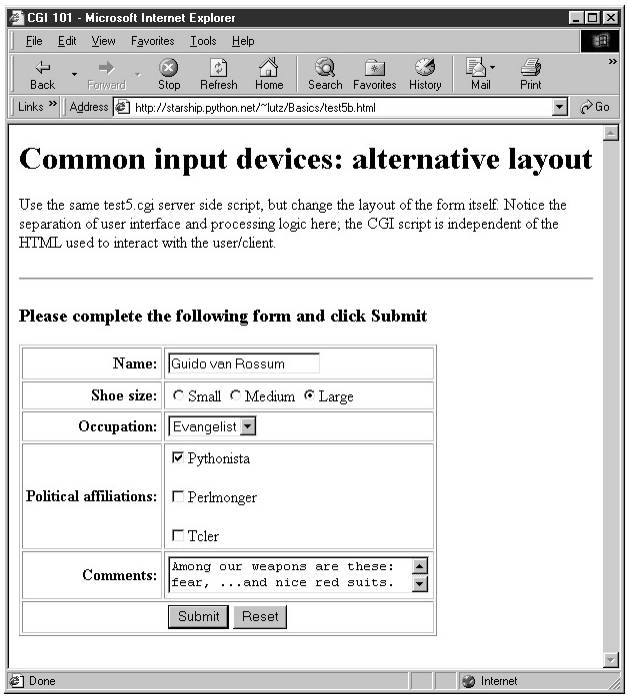


Рис. 12.15. Страница формы, созданная test5b.html

Теперь, когда вы глядите во все глаза, пытаясь найти отличие между этим файлом HTML и предыдущим, я должен отметить, что отличия в HTML, порождающем эту страницу, значительно менее важны, чем то обстоятельство, что поля `action` в формах этих двух страниц указывают на одинаковые URL. Нажатие кнопки Submit в этой версии запускает тот же самый и совершенно не изменившийся сценарий Python CGI *test5.cgi* (пример 12.13).

Это означает, что сценарии совершенно не зависят от структуры интерфейса пользователя, с помощью которого им отправляется информация. Изменения страницы ответа требуют, конечно, изменения сценария, но HTML-код страницы ввода можно изменять по своему вкусу, что не оказывает влияния на код Python, выполняемый на сервере. Рис. 12.16 показывает страницу ответа, порождаемую сценарием на этот раз.

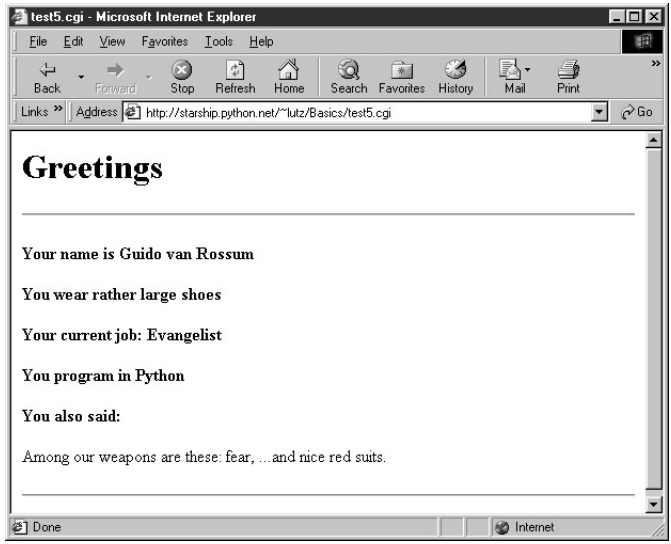


Рис. 12.16. Страница ответа, созданная test5.cgi (2)

Передача параметров в жестко закодированных URL

Ранее мы передавали параметры сценариям CGI, указывая их в конце URL, вводимого в поле адреса броузера (после `?`). Но ничего священного в поле адреса броузера нет. В частности, ничто не может помешать нам использовать тот же синтаксис URL в гиперссылках, закодированных в определениях веб-страниц. Например, веб-страница примера 12.15 определяет три гиперссылки (текст между тегами `<A>` и `</A>`), каждая из которых снова запускает наш первоначальный сценарий *test5.cgi*, но с тремя предварительно закодированными наборами параметров.

Пример 12.15. PP2E\Internet\Cgi-Web\Basics\test5c.html

```
<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices: URL parameters</H1>

<P>This demo invokes the test5.cgi server-side script again, but hardcodes input data
to the end of the script's URL, within a simple hyperlink (instead of packaging
up a form's inputs).  Click your browser's "show page source" button
to view the links associated with each list item below.
```

```
<P>This is really more about CGI than Python, but notice that Python's cgi
module handles both this form of input (which is also produced by GET
form actions), as well as POST-ed forms; they look the same
to the Python CGI script.  In other words, cgi module users are independent
of the method used to submit data.

<P>Also notice that URLs with appended input values like this can be generated
as part of the page output by another CGI script, to direct a next user click
to the right place and context; together with type 'hidden' input fields,
they provide one way to save state between clicks.
</P><HR>

<UL>
<LI><A href="test5.cgi?name=Bob&shoesize=small">Send Bob, small</A>
<LI><A href="test5.cgi?name=Tom&language=Python">Send Tom, Python</A>
<LI><A href=
"http://starship.python.net/~lutz/Basics/test5.cgi?job=Evangelist&comment=spam">
Send Evangelist, spam</A>
</UL>

<HR></BODY></HTML>
```

Этот статический файл HTML определяет три гиперссылки – первые две минимальные, а третья задана полностью, но все они действуют аналогичным образом (опять-таки, целевому сценарию это безразлично). При переходе по URL этого файла отображается страница, показанная на рис. 12.17. В основном это просто страница для запуска готовых вызовов сценария CGI.

При щелчке по второй ссылке этой страницы создается страница ответа, показанная на рис. 12.18. Эта ссылка вызывает сценарий CGI с параметром `name` равным «Tom» и параметром `language` равным «Python», поскольку эти параметры и их значения жест-

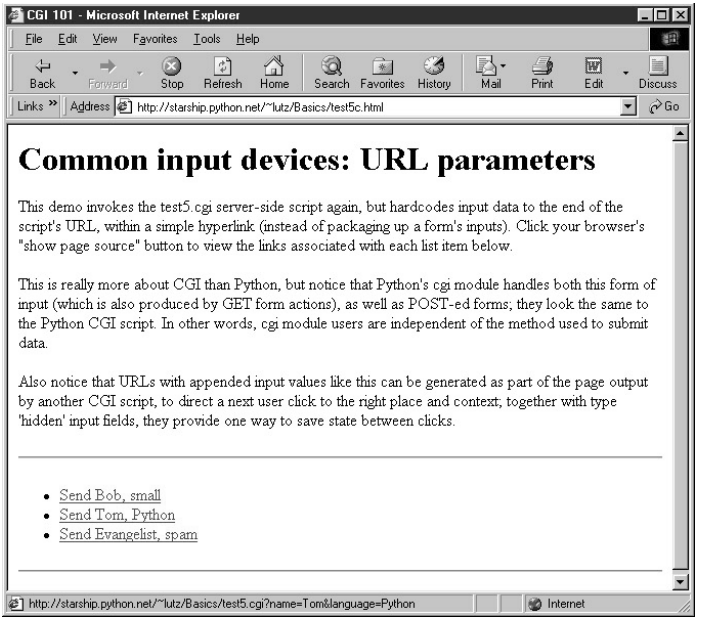


Рис. 12.17. Страница гиперссылок, созданная test5c.html

ко закодированы в URL, указанном в HTML для второй гиперссылки. Это в точности то же самое, как если бы мы вручную ввели строку, показанную вверху броузера на рис. 12.18.

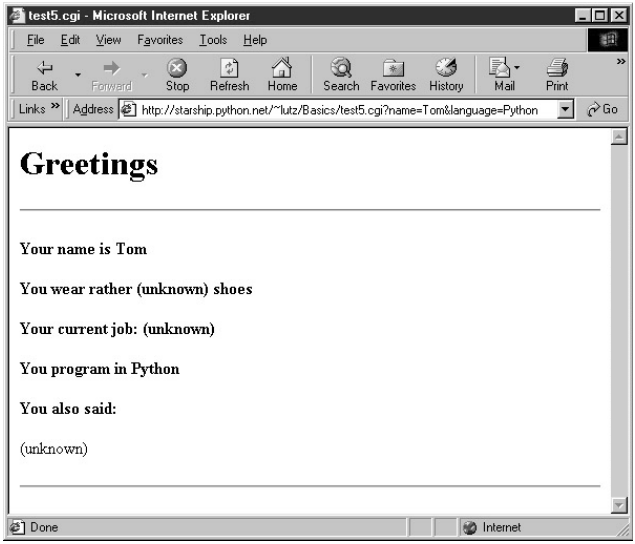


Рис. 12.18. Страница ответа, созданная test5.cgi (3)

Обратите внимание, что здесь отсутствуют многие поля. Сценарий *test5.cgi* сообразит и обнаружит отсутствующие поля, выдав для них сообщение `unknown` на странице ответа. Следует также подчеркнуть, что здесь мы опять повторно используем CGI-сценарий Python. Сам сценарий совершенно не зависит от формата интерфейса пользователя для страницы формы и способа, которым он вызван (при передаче формы или переходе по жестко закодированному URL). В результате разделения интерфейса пользователя и логики обработки сценарии CGI становятся многократно используемыми программными компонентами – по крайней мере, в контексте окружения CGI.

Сохранение информации о состоянии сценария CGI

Но реальной причиной показа этого приема является то, что мы собираемся интенсивно использовать его в более крупных практических примерах последующих двух глав для реализации списков динамически генерируемых выборов, которые «знают», что нужно делать при щелчке по ним. Заранее кодируемые параметры в URL дают возможность сохранять информацию о состоянии при переходе с одной страницы на другую – с их помощью можно управлять действием следующего выполняемого сценария. В этом качестве гиперссылки с такими параметрами иногда называют «умными ссылками» (smart links).

Обычно сценарии CGI выполняются автономно, не имея представления о том, какие сценарии выполнялись перед ними. До сих пор это не имело значения в наших примерах, но более крупные системы обычно состоят из нескольких этапов взаимодействия с пользователем и многих сценариев, поэтому есть потребность в слежении за собираемой попутно информацией. Генерация жестко заданных URL с параметрами предоставляет сценариям CGI одну из возможностей передачи данных очередному сценарию в приложении. При щелчке по ссылкам с такими URL параметры передают запрограммированную информацию о выборе другому сценарию обработки на стороне сервера.

Например, сайт, позволяющий читать электронную почту, может показывать список сообщений для просмотра, реализованный на HTML как список гиперссылок, созданных другим сценарием. В каждую гиперссылку можно включить имя сценария для просмотра сообщений вместе с параметрами, указывающими номер выбранного сообщения, имя почтового сервера и т. д. – те данные, которые требуются для загрузки сообщения, связанного с конкретной ссылкой. Сайт электронной торговли может выдавать сгенерированный список ссылок на товары, каждая из которых запускает жестко закодированную гиперссылку, содержащую номер товара, его цену и т. д.

В целом есть ряд способов передачи или сохранения информации о состоянии при запуске сценариев CGI:

- *Жестко закодированные параметры URL* в динамически генерируемых гиперссылках, встраиваемых в веб-страницы (о чем здесь говорилось)
- *Скрытые (hidden) поля ввода форм*, прикрепляемые к данным формы и встроенные в веб-страницы, но не отображаемые при показе веб-страниц
- *«cookies» HTTP*, хранящиеся на машине клиента и передающиеся между клиентом и сервером в заголовках сообщений HTTP
- *Общие хранилища информации на серверах*, включая базы данных, хранилища постоянных объектов, плоские файлы и т. д.

С большинством из этих способов мы встретимся в дальнейших примерах этой и двух последующих глав.

Селектор «Hello World»

Настало время заняться чем-нибудь более полезным (или хотя бы более увлекательным). В этом разделе представлена программа, которая показывает базовый синтаксис, требуемый различными языками для вывода строки «Hello World», классической начальной точки. Для простоты предполагается вывод строки в стандартный выходной поток, а не в GUI или на веб-странице. Кроме того, показывается лишь сама команда вывода, а не полная программа. Версия Python оказывается при этом завершенной программой, но мы не станем сейчас противопоставлять это конкурирующим языкам.

Структурно этот пример в первом приближении состоит из файла HTML главной страницы и написанного на Python CGI-сценария, запускаемого из формы на главной странице HTML. Никакого состояния или данных в базе при щелчках пользователя не сохраняется, поэтому данный пример все еще достаточно прост. На самом деле главная страница HTML, реализуемая примером 12.16, является просто одним большим выпадающим окном списка выбора, находящимся внутри формы.

Пример 12.16. PP2E\Internet\Cgi-Web\Basics\languages.html

```
<html><body>
<title>Languages</title>
<h1>Hello World selector</h1>

<P>This demo shows how to display a "hello world" message in various
programming languages' syntax.  To keep this simple, only the output command
is shown (it takes more code to make a complete program in some of these
languages), and only text-based solutions are given (no GUI or HTML
construction logic is included). This page is a simple HTML file; the one
you see after pressing the button below is generated by a Python CGI script
which runs on the server. Pointers:

<UL>
```

```
<LI>To see this page's HTML, use the 'View Source' command in your browser.
<LI>To view the Python CGI script on the server,
    <A HREF="languages-src.cgi">click here</A> or
    <A HREF="getfile.cgi?filename=languages.cgi">here</A>.
<LI>To see an alternative version that generates this page dynamically,
    <A HREF="languages2.cgi">click here</A>.
<LI>For more syntax comparisons, visit
    <A HREF="http://www.ionet.net/~timtroyr/funhouse/beer.html">this site</A>.
</UL></P>

<hr>
<form method=POST action="languages.cgi">
    <P><B>Select a programming language:</B>
    <P><select name=language>
        <option>All
        <option>Python
        <option>Perl
        <option>Tcl
        <option>Scheme
        <option>SmallTalk
        <option>Java
        <option>C
        <option>C++
        <option>Basic
        <option>Fortran
        <option>Pascal
        <option>Other
    </select>
    <P><input type=Submit>
</form>

</body></html>
```

Не будем пока обращать внимание на некоторые гиперссылки в середине этого файла: они приводят нас к таким большим понятиям, как передача файлов и сопровождаемость, которыми мы займемся в следующих двух разделах. При открытии этого файла HTML в броузере клиента выводится страница, представленная на рис. 12.19.

Графический элемент, находящийся над кнопкой Submit, является выпадающим окном списка, позволяющим выбрать одно из значений тега `<option>` в файле HTML. Как обычно, выбрав название одного из языков и нажав кнопку Submit внизу (или клавишу <Enter>), мы отправим название выбранного языка экземпляру программы сценария CGI на сервере, имя которой указано в параметре формы `action`. В примере 12.17 приведен сценарий Python, выполняемый на сервере в результате передачи.

Пример 12.17. PP2E\Internet\Cgi-Web\Basics\languages.cgi

```
#!/usr/bin/python
###############################################################################################
# Показать синтаксис "hello world" для заданного языка; обратите внимание на необрабатываемые
# строки r'...', позволяющие сохранить '\n' в таблице, и cgi.escape() над строкой,
# чтобы символы типа '<<' не смущали броузеры - они транслируются в допустимый код html;
# в сценарий можно передать любое имя языка: например, можно ввести
# "http://starship.python.net/~lutz/Basics/languages.cgi?language=Cobol"  в любом
# веб-броузере. Предупреждение: список языков есть в обоих файлах - cgi и html, можно было
# бы импортировать его из одного файла, если генерировать список другим cgi-сценарием;
###############################################################################################

debugme  = 0                                    # 1=тест из командной строки
```

```
inputkey = 'language'                              # имя входного параметра

hellos = {
    'Python':    r" print 'Hello World'              ",
    'Perl':      r' print "Hello World\n";           ',
    'Tcl':       r' puts "Hello World"               ',
    'Scheme':    r' (display "Hello World") (newline) ',
    'SmallTalk': r" 'Hello World' print.            ",
    'Java':      r' System.out.println("Hello World"); ',
    'C':         r' printf("Hello World\n");        ',
    'C++':       r' cout << "Hello World" << endl;  ',
    'Basic':     r' 10 PRINT "Hello World"          ',
    'Fortran':   r" print *, 'Hello World'           ",
    'Pascal':    r" WriteLn('Hello World');          "
}

class dummy:                                       # имитация объекта ввода
    def __init__(self, str): self.value = str

import cgi, sys
if debugme:
    form = {inputkey: dummy(sys.argv[1])}          # имя в командной строке
else:
    form = cgi.FieldStorage()                      # разбор реальных данных

print "Content-type: text/html\n"                  # добавить пустую строку
print "<TITLE>Languages</TITLE>"
print "<H1>Syntax</H1><HR>"

def showHello(form):                               # html для одного языка
    choice = form[inputkey].value
    print "<H3>%s</H3><P><PRE>" % choice
```

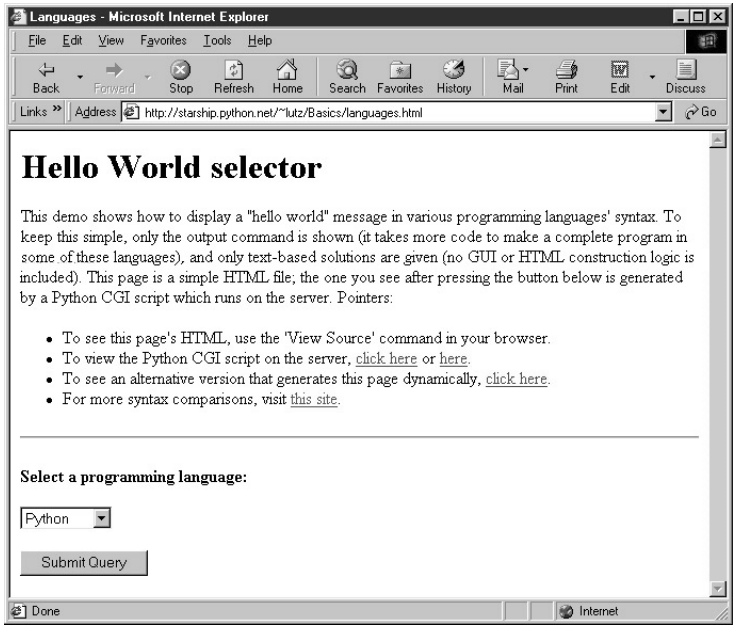


Рис. 12.19. Главная страница «Hello World»

```
        try:
            print cgi.escape(hellos[choice])
        except KeyError:
            print "Sorry--I don't know that language"
        print "</PRE></P><BR>"

    if not form.has_key(inputkey) or form[inputkey].value == 'All':
        for lang in hellos.keys():
            mock = {inputkey: dummy(lang)}
            showHello(mock)
    else:
        showHello(form)
    print '<HR>'
```

Как обычно, этот сценарий выводит код HTML в стандартный выходной поток, чтобы создать страницу ответа в броузере клиента. В этом сценарии немного нового, о чем можно сказать, но в нем использованы некоторые приемы, заслуживающие особого внимания:

Необрабатываемые строки

Обратите внимание на использование *необрабатываемых строк* (raw strings, константы строк, которым предшествует символ «r») в синтаксисе словаря языков. Вспомните, что необрабатываемые строки сохраняют в строке символы обратной косой черты \ буквально, не интерпретируя их как начало escape-кодов. Без этого последовательность символов перевода строки `\n` в фрагментах кода некоторых языков интерпретировалась бы Python как перевод строки, а не выводилась в ответный HTML как `\n`.

Преобразование текста, встраиваемого в HTML и URL

Этот сценарий следит за тем, чтобы текст фрагмента кода на каждом языке форматировался посредством вспомогательной функции `cgi.escape`. Эта стандартная утилита Python автоматически транслирует специальные символы HTML в последовательности escape-кодов HTML, чтобы броузеры не обрабатывали их как операторы HTML. Технически `cgi.escape` выполняет трансляцию в последовательности escape-кодов в соответствии со стандартным соглашением HTML: `<`, `>` и `&` превращаются в `<`, `>` и `&`. Если передать `true` в качестве второго аргумента, то и символ двойной кавычки (") транслируется в `&quot;`.

Например, оператор перенаправления в поток `<<` в коде C++ транслируется в `<<` – пару escape-кодов HTML. Поскольку каждый выводимый фрагмент кода в итоге встраивается в ответный поток HTML, необходимо преобразовывать все содержащиеся в нем специальные символы HTML. Анализаторы HTML (в том числе стандартный модуль Python `htmllib`) транслируют escape-коды обратно в исходные символы при отображении страницы.

В более широком смысле, поскольку CGI основывается на концепции передачи форматированных строк через Сеть, операция преобразования специальных символов встречается повсеместно. Сценариям CGI почти всегда для надежности приходится преобразовывать текст, генерируемый для включения в ответ. Например, если возвращается произвольный текст, введенный пользователем или полученный из источника данных на сервере, обычно нельзя гарантировать, что в нем не будет специальных символов HTML, поэтому на всякий случай нужно его преобразовать.

В последующих примерах мы также увидим, что символы, вставляемые в строки адресов URL, генерируемых нашими сценариями, тоже могут потребовать преобразования. Например, литеральный `&` в URL является специальным и должен быть преобразован, если появляется в тексте, вставляемом в URL. Однако синтак-

сис URL резервирует другие специальные символы в сравнении с кодом HTML, а потому должны использоваться другие инструменты и соглашения по преобразованию. Как мы увидим далее в этой главе, `cgi.escape` реализует трансляцию в коде HTML, а `urllib.quote` (и родственные) преобразует символы в строках URL.

Имитация данных, вводимых через форму

Здесь снова ввод данных из формы имитируется (моделируется) как для целей отладки, так и для ответа на запрос всех языков в списке. Если глобальная переменная сценария `debugme` имеет значение «истина», к примеру, то сценарий создает словарь, взаимозаменяемый с результатом вызова `cgi.FieldStorage` – его ключ «языка» ссылается на экземпляр подставного класса `dummy`. Этот класс, в свою очередь, создает объект с таким же интерфейсом, как в содержимом результата `cgi.FieldStorage` – он строит объект, для которого атрибут `value` установлен равным переданной строке.

В итоге мы можем протестировать этот сценарий, запустив его из командной строки системы: сгенерированный словарь заставляет сценарий считать, что его вызвал броузер через Сеть. Аналогично, если названием запрашиваемого языка является «All», сценарий обходит все записи в таблице языков, создавая из них искусственный словарь (как если бы пользователь поочередно запросил все языки). Это позволяет повторно использовать имеющуюся логику `showHello` и отобразить код всех языков на одной странице. Как всегда в Python, мы пишем код для интерфейсов объектов и протоколов, а не конкретных типов данных. Функция `showHello` успешно обработает любой объект, отвечающий на синтаксис `form['language'].value`.[1]

Теперь снова вернемся к взаимодействию с этой программой. Если выбрать конкретный язык, наш сценарий CGI генерирует в ответ HTML такого типа (а также необходимые заголовок content-type и пустую строку):

```
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Scheme</H3><P><PRE>
 (display "Hello World") (newline)
</PRE></P><BR>
<HR>
```

Код программы помечен тегом `<PRE>`, задающим преформатированный текст (броузер не станет форматировать его как абзац обычного текста). Этот код ответа показывает, что мы получаем при выборе «Scheme». Рис. 12.20 показывает страницу, выдаваемую сценарием после выбора «Python» в выпадающем списке.

Наш сценарий принимает также в качестве имени языка «All» и интерпретирует это как запрос вывода синтаксиса для всех известных языков. Например, вот HTML, который генерируется, если установить глобальную переменную `debugme` равной 1 и запустить сценарий из командной строки с одним аргументом «All». Это тот же вывод, который появится в броузере клиента в ответ на запрос «All»:[2]

---

[1] Если вы внимательно читаете, то могли заметить, что мы уже второй раз использовали моделирование в этой главе (см. выше пример *test4.cgi*). Если вы считаете этот прием полезным вообще, может иметь смысл поместить класс `dummy` и функцию для заполнения словаря формы по требованию в отдельный модуль, который можно многократно использовать. На самом деле мы и сделаем это в следующем разделе. Даже для таких классов-двустрочников ввод одного и того же кода третий раз подряд вполне может убедить в мощи повторного использования кода.

[2] Интересно, что мы также получим ответ «All», если установить `debugme` в 0 при запуске сценария из командной строки. `cgi.FieldStorage` возвращает пустой словарь при вызове не из окружения CGI, не возбуждая исключительной ситуации, поэтому выполняется проверка отсутствия ключа. Однако вряд ли следует полагаться на такое поведение.

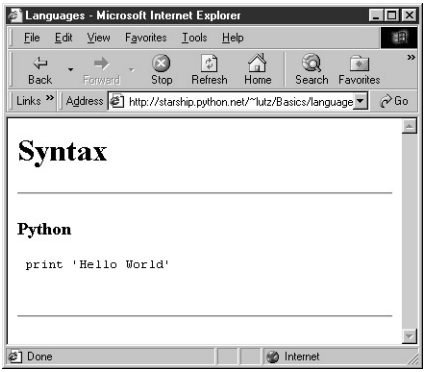


Рис. 12.20. Страница ответа, создаваемая languages.cgi

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python languages.cgi All
Content-type: text/html

<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Perl</H3><P><PRE>
 print "Hello World\n";
</PRE></P><BR>
<H3>SmallTalk</H3><P><PRE>
 'Hello World' print.
</PRE></P><BR>
<H3>Basic</H3><P><PRE>
 10 PRINT "Hello World"
</PRE></P><BR>
<H3>Scheme</H3><P><PRE>
 (display "Hello World") (newline)
</PRE></P><BR>
<H3>Python</H3><P><PRE>
 print 'Hello World'
</PRE></P><BR>
<H3>C++</H3><P><PRE>
 cout &lt;&lt; "Hello World" &lt;&lt; endl;
</PRE></P><BR>
<H3>Pascal</H3><P><PRE>
 WriteLn('Hello World');
</PRE></P><BR>
<H3>Java</H3><P><PRE>
 System.out.println("Hello World");
</PRE></P><BR>
<H3>C</H3><P><PRE>
 printf("Hello World\n");
</PRE></P><BR>
<H3>Tcl</H3><P><PRE>
 puts "Hello World"
</PRE></P><BR>
<H3>Fortran</H3><P><PRE>
 print *, 'Hello World'
</PRE></P><BR>
<HR>
```

Все языки представлены здесь по одной и той же схеме кода – функция `showHello` вызывается для каждой записи в таблице вместе с моделируемым объектом формы. Обратите внимание на то, как преобразуется код C++ для встраивания в поток HTML; это работа вызова `cgi.escape`. При просмотре в броузере страница ответа «All» выводится, как показано на рис. 12.21.

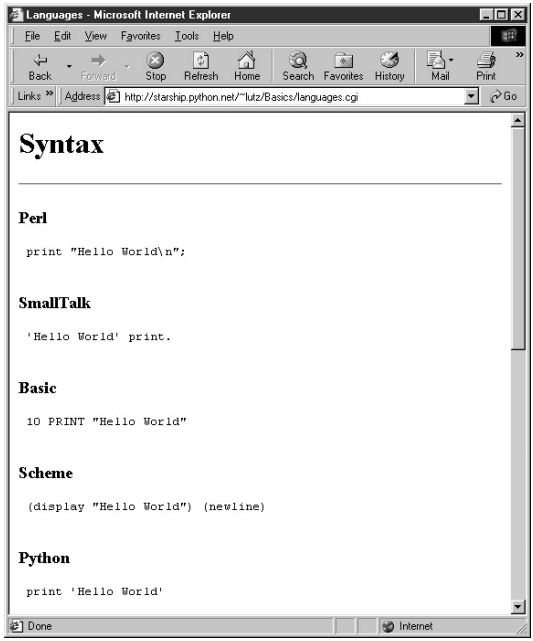


Рис. 12.21. Страница ответа для выбора «all languages»

Проверка отсутствующих или недопустимых данных

До сих пор мы запускали этот сценарий CGI, выбирая название языка из выпадающего списка на главной странице HTML. В данном контексте можно быть вполне уверенным, что сценарий получит допустимые входные данные. Однако обратите внимание, что ничто не мешает пользователю передать запрашиваемое название языка в конце URL сценария CGI в качестве явно заданного параметра, не используя форму страницы HTML. Например, URL вида:

```
http://starship.python.net/~lutz/Basics/languages.cgi?language=Python
```

дает ту же страницу ответа «Python», которая показана на рис. 12.20.[1] Однако, поскольку пользователь всегда может обойти файл HTML и использовать явный URL, существует возможность вызвать наш сценарий с названием неизвестного языка, которого нет в выпадающем списке файла HTML (и потому в таблице нашего сценария).

[1] См. в примерах с модулем `urllib` в предыдущей и последующих главах способ отправки этого URL из сценария Python. `urllib` позволяет программам загружать веб-страницы и вызывать удаленные сценарии CGI путем создания и передачи строк URL типа этой, в которых необходимые параметры помещаются в конец строки. Можно, например, воспользоваться этим модулем, чтобы автоматически отправлять информацию для заказа книг по Python в магазине, торгующем через Интернет, из сценария Python, не прибегая к запуску веб-броузера.

На самом деле сценарий можно запустить вообще без языка, если явно набрать его URL без параметров.

Для надежности сценарий явно проверяет оба случая, как должны в целом делать все сценарии CGI. Например, вот HTML, сгенерированный в ответ на запрос фиктивного языка «GuiDO»:

```
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>GuiDO</H3><P><PRE>
Sorry--I don't know that language
</PRE></P><BR>
<HR>
```

Если сценарий не получает во входных данных названия языка, по умолчанию обрабатывается вариант «All». Если не обнаруживать такие случаи, то весьма возможной была бы тихая смерть сценария по исключительной ситуации Python с оставлением у пользователя по большей части бесполезной незавершенной страницы или страницы по умолчанию сообщения об ошибке (здесь мы не назначали `stderr` в `stdout`, поэтому сообщение Python об ошибке не выводилось бы). Наглядно это показано на рис. 12.22, где приведена страница, генерируемая при вызове такого явно заданного URL:

```
http://starship.python.net/~lutz/Basics/languages.cgi?language=COBOL
```

Для проверки такого варианта ошибки в выпадающий список включено название «Unknown», создающее аналогичную ответную страницу ошибки. Добавление кода в таблицу сценария для программы «Hello World» на COBOL оставляется читателю в качестве упражнения.

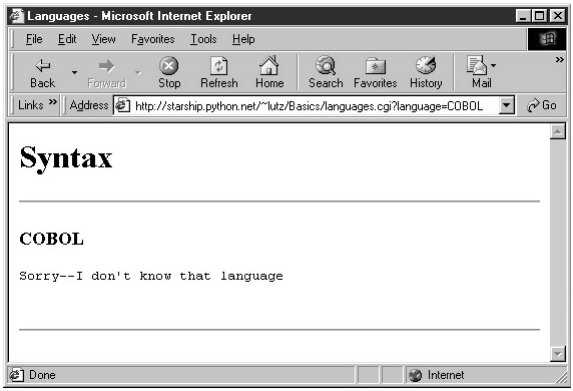


Рис. 12.22. Страница ответа для неизвестного языка

Код, облегчающий сопровождение

Отойдем на некоторое время от деталей написания кода, чтобы взглянуть на него с точки зрения архитектуры. Как мы видели, код Python в целом автоматически образует системы, которые легко читать и сопровождать: у него простой синтаксис, в значительной мере освобождающий от нагромождений, создаваемых другими инструментами. С другой стороны, стиль программирования и архитектура программы часто могут не меньше, чем синтаксис, влиять на легкость сопровождения. Например, страницы селектора «Hello World» выше в этой главе выполняют требуемую функ-

цию, и «слепить» их удалось быстро и легко. Однако в существующем виде этот селектор языков имеет существенные недостатки в отношении эффективности сопровождения.

Представим себе, например, что вы действительно возьметесь за задачу, которую я предложил в конце предыдущего раздела, и попробуете добавить еще одну запись для COBOL. Если добавить COBOL в таблицу сценария CGI, это будет лишь полрешения: список поддерживаемых языков избыточно размещен в двух местах – в HTML для главной страницы и в словаре синтаксиса сценария. При изменении одного из них другой не меняется. В более широком случае эта программа не прошла бы придирчивого контроля качества кода в целом ряде отношений:

Список выбора

Как только что было отмечено, список языков, которые поддерживает эта программа, находится в двух местах – в файле HTML и таблице сценария CGI.

Имя поля

Имя поля входного параметра, «language», жестко закодировано в обоих файлах. Если вы измените его в одном файле, то можете и не вспомнить, что нужно изменить его также в другом.

Имитация формы

В этой главе мы уже кодировали лишний второй раз класс для имитации ввода из полей формы, поэтому класс «dummy» явно нужно сделать повторно используемым механизмом.

Код HTML

Встроенный в сценарий и генерируемый им HTML разбросан по всей программе в командах `print`, из-за чего трудно осуществлять значительные изменения структуры веб-страницы.

Конечно, это небольшой пример, но проблемы избыточности и повторного использования встают более остро по мере роста размеров сценариев. Следует придерживаться практического правила, что если для изменения какой-то одной особенности поведения приходится редактировать несколько исходных файлов или при написании программ используется копирование через буфер частей уже существующего кода, то надо подумать о рационализации структуры программы. Чтобы проиллюстрировать стиль и приемы программирования, более дружественные к сопровождению, перепишем наш пример и исправим сразу все отмеченные слабости.

Шаг 1: совместное использование объектов разными страницами

Первые две из перечисленных выше проблем сопровождения можно снять с помощью простого преобразования; решение заключается в динамической генерации главной страницы выполняемым сценарием вместо использования готового файла HTML. Сценарий может импортировать имя поля ввода и значения для списка выбора из общего файла модуля Python, совместно используемого сценариями генерации главной страницы и страницы ответа. При изменении списка выбора или имени поля в общем модуле автоматически изменяются оба клиента. Сначала переместим совместно используемые объекты в файл общего модуля, как показано в примере 12.18.

Пример 12.18. PP2E\Internet\Cgi-Web\Basics\languages2common.py

```
##############################################################################
# Общие объекты, совместно используемые сценариями главной и ответной страниц;
```

```
# для добавления нового языка нужно изменить только эту страницу.
################################################################################

inputkey = 'language'                           # имя входного параметра

hellos = {
    'Python':     r" print 'Hello World'               ",
    'Perl':       r' print "Hello World\n";            ',
    'Tcl':        r' puts "Hello World"                ',
    'Scheme':     r' (display "Hello World") (newline) ',
    'SmallTalk':  r" 'Hello World' print.              ",
    'Java':       r' System.out.println("Hello World"); ',
    'C':          r' printf("Hello World\n");          ',
    'C++':        r' cout << "Hello World" << endl;    ',
    'Basic':      r' 10 PRINT "Hello World"            ',
    'Fortran':    r" print *, 'Hello World'             ",
    'Pascal':     r" WriteLn('Hello World');            "
}
```

В модуле `languages2common` **содержатся все данные, которые должны быть согласованы на страницах: имя поля и словарь синтаксиса. Словарь синтаксиса** `hellos` **– не совсем тот код HTML, который нужен, но список его ключей можно использовать для динамической генерации HTML списка выбора на главной странице. Далее в примере 12.19 мы перепишем главную страницу в виде выполняемого сценария, помещающего в HTML ответа значения, импортированные из файла общего модуля предыдущего примера.**

Пример 12.19. PP2E\Internet\Cgi-Web\Basics\languages2.cgi

```
#!/usr/bin/python
#############################################################################################
# Динамическая генерация html главной страницы исполняемым сценарием Python вместо готового
# файла HTML; это позволяет импортировать имя ожидаемого поля ввода и значения таблицы
# выбора из файла общего модуля Python; теперь изменения нужно проводить только
# в одном месте, файле модуля Python;
#############################################################################################

REPLY = """Content-type: text/html

<html><body>
<title>Languages2</title>
<h1>Hello World selector</h1>
<P>Similar to file <a href="languages.html">languages.html</a>, but
this page is dynamically generated by a Python CGI script, using
selection list and input field names imported from a common Python
module on the server. Only the common module must be maintained as
new languages are added, because it is shared with the reply script.

To see the code that generates this page and the reply, click
<a href="getfile.cgi?filename=languages2.cgi">here</a>,
<a href="getfile.cgi?filename=languages2reply.cgi">here</a>,
<a href="getfile.cgi?filename=languages2common.py">here</a>, and
<a href="getfile.cgi?filename=formMockup.py">here</a>.</P>
<hr>
<form method=POST action="languages2reply.cgi">
    <P><B>Select a programming language:</B>
    <P><select name=%s>
        <option>All
        %s
        <option>Other
```

```
    </select>
    <P><input type=Submit>
</form>
</body></html>
"""

import string
from languages2common import hellos, inputkey

options = []
for lang in hellos.keys():
    options.append('<option>' + lang)      # поместить ключи таблицы в код html
options = string.join(options, '\n\t')
print REPLY % (inputkey, options)          # имя поля и значения из модуля
```

Пока не обращайте внимания на гиперссылки `getfile` в этом файле, их значение будет объяснено в следующем разделе. Обратите, однако, внимание, что определение страницы HTML стало печатаемой строкой Python (с именем `REPLY`) со спецификаторами форматирования `%s`, которые будут заменены значениями, импортированными из общего модуля.[1] В остальных отношениях оно аналогично коду первоначального файла HTML: при переходе по URL этого сценария выводится сходная страница, показанная на рис. 12.23. Но на этот раз страница создается в результате выполнения на сервере сценария, который заполняет выпадающий список выбора значениями из списка ключей в общей таблице синтаксиса.

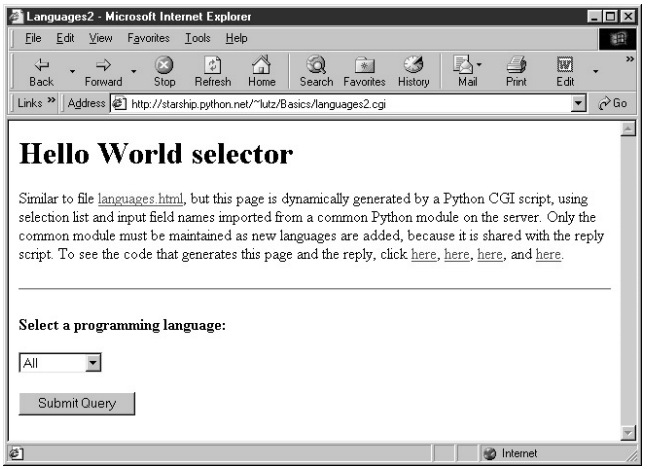


Рис. 12.23. Альтернативная главная страница, созданная languages2.cgi

Шаг 2: многократно используемая утилита имитации формы

Перемещение таблицы языков и имени поля ввода в файл модуля решает первые две отмеченные проблемы сопровождения. Но если мы хотим избавиться от написания

[1] Шаблон кода HTML тоже можно было бы загружать из внешнего текстового файла, но редактировать внешние текстовые файлы ничуть не проще, чем сценарии Python. На самом деле сценарии Python и есть текстовые файлы, и это важная особенность языка: обычно легко можно изменить сценарии Python работающей системы прямо на месте, без повторной компиляции или компоновки.

фиктивного класса, имитирующего поля, в каждом создаваемом сценарии CGI необходимо сделать еще кое-что. И снова для этого нужно просто воспользоваться привлекательностью модулей Python для повторного использования кода: переместим фиктивный класс во вспомогательный модуль, показанный в примере 12.20.

Пример 12.20. PP2E\Internet\Cgi-Web\Basics\formMockup.py

```
##################################################################
# Средства моделирования результата вызова a cgi.FieldStorage();
# полезны для тестирования сценариев CGI вне веб
##################################################################

import types

class FieldMockup:                              # имитируемый объект ввода
    def __init__(self, str):
        self.value = str

def formMockup(**kwargs):                       # передать аргументы поле=значение
    mockup = {}                                 # множественный выбор: [value,...]
    for (key, value) in kwargs.items():
        if type(value) is not types.ListType:   # у простых полей есть .value
            mockup[key] = FieldMockup(str(value))
        else:                                   # для множественного выбора есть список
            mockup[key] = []                    # доделать: поля отправки файлов
            for pick in value:
                mockup[key].append(FieldMockup(pick))
    return mockup

def selftest():
    # такая форма, если поля можно жестко закодировать
    form = formMockup(name='Bob', job='hacker', food=['Spam', 'eggs', 'ham'])
    print form['name'].value
    print form['job'].value
    for item in form['food']:
        print item.value,
    # использовать реальный словарь, если ключи в переменных или вычисляются
    print
    form = {'name':FieldMockup('Brian'), 'age':FieldMockup(38)}
    for key in form.keys():
        print form[key].value

if __name__ == '__main__': selftest()
```

В результате помещения имитирующего класса в этот модуль *formMockup.py* он автоматически становится многократно используемым инструментом и может быть импортирован в любой сценарий, который мы соберемся написать.[1] Для лучшей читаемости класс моделирования поля `dummy` переименован в `FieldMockup`. Для удобства введена также вспомогательная функция `formMockup`, которая строит весь словарь формы соответственно переданным ей аргументам ключевых слов. Предполагая, что имена подделываемой формы могут быть жестко закодированы, модель создается за один вызов. В модуле есть также функция самотестирования, вызываемая при запуске этого файла из командной строки и демонстрирующая использование его экспорта. Вот

[1] При этом, конечно, предполагается, что при выполнении этих сценариев этот модуль находится в пути поиска Python. О пути поиска читайте выше в этой главе. Так как Python по умолчанию ведет поиск импортируемых модулей в текущем каталоге, это всегда выполняется без редактирования `sys.path`, если все файлы находятся в главном веб-каталоге.

этот контрольный вывод, образуемый путем создания и опроса двух объектов моделирования формы:

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python formMockup.py
Bob
hacker
Spam eggs ham
38
Brian
```

Так как теперь имитация находится в модуле, можно воспользоваться ею всякий раз, когда нужно протестировать сценарий CGI в автономном режиме. Для иллюстрации в примере 12.21 приводится переработка ранее встречавшегося *test5.cgi*, в котором утилита имитации формы использована для моделирования полей ввода. Если бы мы спланировали заранее, то таким способом могли бы проверить этот сценарий безо всякого подключения к Сети.

Пример 12.21. PP2E\Internet\Cgi-Web\Basics\test5_mockup.cgi

```
#!/usr/bin/python
########################################################################
# Выполнить логику test5 с formMockup вместо cgi.FieldStorage()
# для проверки: python test5_mockup.cgi > temp.html и открыть temp.html
########################################################################

from formMockup import formMockup
form = formMockup(name='Bob',
                  shoesize='Small',
                  language=['Python', 'C++', 'HTML'],
                  comment='ni, Ni, NI')

# остальное, как в оригинале, исключая присвоение form
```

Выполнение этого сценария из простой командной строки показывает, как будет выглядеть ответный поток HTML:

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python test5_mockup.cgi
Content-type: text/html

<TITLE>test5.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<H4>Your name is Bob</H4>
<H4>You wear rather Small shoes</H4>
<H4>Your current job: (unknown)</H4>
<H4>You program in Python and C++ and HTML</H4>
<H4>You also said:</H4>
<P>ni, Ni, NI</P>
<HR>
```

При запуске из броузера будет выведена страница, показанная на рис. 12.24. Значения полей ввода здесь жестко закодированы, что похоже по духу на расширение *test5*, в котором входные параметры встраивались в конец URL гиперссылок. Здесь они поступают от объектов моделирования формы, создаваемых в сценарии ответа, которые можно изменить только редактированием сценария. Однако, поскольку код Python выполняется немедленно, модификация сценария Python во время цикла отладки происходит со скоростью ввода с клавиатуры.

Рис. 12.24. Страница ответа с имитируемыми входными данными

Шаг 3: объединим все вместе – новый сценарий ответа

Остался один последний шаг на пути к нирване сопровождения программного обеспечения: мы еще должны переписать сам сценарий ответной страницы, чтобы импортировать данные, выделенные в общий модуль, и импортировать средства из многократно используемого модуля имитации формы. Занявшись этим делом, поместим код в функции (на случай, если мы когда-нибудь поместим в этот файл то, что захочется импортировать в другой сценарий), а весь код HTML – в блоки строк, заключенные в тройные кавычки (см. пример 12.22). Изменять HTML обычно проще, когда он выделен в такие отдельные строки, а не разбросан по всей программе.

Пример 12.22. PP2E\Internet\Cgi-Web\Basics\languages2reply.cgi

```
#!/usr/bin/python
###############################################################################
# Для облегчения сопровождения использовать строки шаблонов html,
# получить таблицу языков и входной ключ из файла общего модуля,
# получить многократно используемый модуль утилит имитации полей формы.
###############################################################################

import cgi, sys
from formMockup import FieldMockup              # имитатор поля ввода
from languages2common import hellos, inputkey  # получить общую таблицу, имя
debugme = 0

hdrhtml = """Content-type: text/html\n
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>"""

langhtml = """
<H3>%s</H3><P><PRE>
%s
</PRE></P><BR>"""
```

```
def showHello(form):                    # html для одного языка
    choice = form[inputkey].value       # название языка также преобразуется
    try:
        print langhtml % (cgi.escape(choice), cgi.escape(hellos[choice]))
    except KeyError:
        print langhtml % (cgi.escape(choice), "Sorry--I don't know that language")

def main():
    if debugme:
        form = {inputkey: FieldMockup(sys.argv[1])}  # имя в командной строке
    else:
        form = cgi.FieldStorage()                    # разбор реальных входных данных

    print hdrhtml
    if not form.has_key(inputkey) or form[inputkey].value == 'All':
        for lang in hellos.keys():
            mock = {inputkey: FieldMockup(lang)}
            showHello(mock)
    else:
        showHello(form)
    print '<HR>'

if __name__ == '__main__': main()
```

Если глобальная переменная `debugme` установлена в 1, сценарий можно тестировать просто из командной строки:

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python languages2reply.cgi Python
Content-type: text/html

<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>

<H3>Python</H3><P><PRE>
 print 'Hello World'
</PRE></P><BR>
<HR>
```

При выполнении в режиме онлайн получаются те же страницы, которые мы видели для первоначальной версии этого примера (не станем повторять их снова). Наше преобразование изменило архитектуру программы, но не ее интерфейс пользователя.

Большая часть изменений, внесенных в эту версию кода сценария ответа, проста. При практическом испытании этих страниц единственными отличиями, которые вы обнаружите, будут URL вверху броузера (все-таки, это разные файлы), дополнительные пустые строки в генерируемом HTML (игнорируются броузером) и, возможно, другой порядок названий языков в выпадающем списке главной страницы.

Это различие в порядке элементов списка выбора возникает потому, что эта версия зависит от порядка в списке ключей словаря Python, а не от фиксированного списка в файле HTML. Словари, если вы помните, упорядочивают свои записи так, чтобы скорейшим образом осуществлять выборку; если вы хотите, чтобы список выбора был более предсказуем, просто отсортируйте список ключей с помощью метода `sort`, прежде чем просматривать его.

Снова об escape-преобразованиях HTML и URL

Возможно, самое тонкое изменение в версии из предыдущего раздела состоит в том, что для надежности в этой редакции также вызывается `cgi.escape` для *названия* языка, а не

Имитация ввода с помощью переменных оболочки

Если вы понимаете свои действия, то иногда можете также протестировать сценарии CGI из командной строки, установив переменные окружения так же, как это делают серверы HTTP, и затем запустив свой сценарий. Например, можно прикинуться веб-сервером, записав входные параметры в переменную окружения QUERY_STRING с помощью того же синтаксиса, который используется в конце строки URL после ?:

```
$ setenv QUERY_STRING "name=Mel&job=trainer,+writer"
$ python test5.cgi
Content-type: text/html
<TITLE>test5.cgi<?TITLE>
<H1>Greetings</H1>
<HR>
<H4>Your name is Mel</H4>
<H4>You wear rather (unknown) shoes</H4>
<H4>Your current job: trainer, writer</H4>
<H4>You program in (unknown)</H4>
<H4>You also said:</H4>
<P>(unknown)</P>
<HR>
```

Здесь мы имитируем действие подачи формы в стиле GET или явного URL. Серверы HTTP помещают строку запроса (параметры) в переменную оболочки QUERY_STRING. Модуль Python `cgi` обнаруживает их там, как если бы они были переданы броузером. Передачу данных в стиле POST тоже можно имитировать с помощью переменных оболочки, но это сложнее, и лучше всего не пытаться узнать, как это сделать. В действительности имитация ввода с помощью объектов Python (как в *formMockup.py*) должна работать надежнее. Но для некоторых сценариев CGI могут существовать дополнительные ограничения окружения или тестирования, требующие особой обработки.

только для фрагмента его кода. Маловероятно, но все же возможно, что некто передаст сценарию название языка, содержащее символ HTML. Например, следующий URL:

```
http://starship.python.net/~lutz/Basics/languages2reply.cgi?language=a<b
```

вставляет < в параметр названия языка (названием является `a<b`). При передаче такого параметра эта версия с помощью `cgi.escape` правильно транслирует < для использования в ответном HTML согласно обсуждавшимся выше стандартным соглашениям HTML:

```
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>

<H3>a&lt;b</H3><P><PRE>
Sorry--I don't know that language
</PRE></P><BR>
<HR>
```

В исходной версии название языка не преобразуется, поэтому встроенный `<b` интерпретируется как тег HTML (в результате чего оставшаяся часть страницы может быть выведена полужирным шрифтом!). Как вы уже можете теперь судить, преобразование текста вездесуще в сценариях CGI – даже текст, который кажется вам безопасным, обычно должен преобразовываться в escape-последовательности перед вставкой в код HTML в ответном потоке.

Соглашения по escape-кодам в URL

Обратите, однако, внимание, что хотя непреобразованные < нельзя вставлять в ответный код HTML, совершенно допустимо включать их дословно в прежние строки URL, используемые для получения ответа. На самом деле в HTML и URL в качестве специальных определены совершенно различные символы. Например, символ & должен быть преобразован в &amp в коде HTML, но для представления литерального & в строке URL должна использоваться совсем другая схема кодировки (там этот символ обычно разделяет параметры). Для передачи сценарию названия языка вроде a&b нужно ввести такой URL:

```
http://starship.python.net/~lutz/Basics/languages2reply.cgi?language=a%26b
```

Здесь %26 представляет & – символ & заменяется % с последующей шестнадцатеричной величиной (0×26) его значения в коде ASCII (38). По стандартам URL большинство не буквенно-цифровых символов должно транслироваться в такие escape-последовательности, а пробелы заменяются знаками +. Технически это соглашение называют форматом строки запроса *application/x-www-form-urlencoded*, и оно стоит за теми странными URL, которые вы можете часто видеть вверху броузера, путешествуя по веб.

Средства Python для кодировки HTML и URL

Вероятно, в вашей памяти, как и в моей, не запечатлелось шестнадцатеричное значение кода ASCII для &. К счастью, в Python есть средства для автоматического преобразования URL, подобные cgi.escape для преобразования HTML. Главное, о чем нужно помнить, это то, что код HTML и строки URL имеют совершенно разный синтаксис и потому используют различные соглашения по преобразованию в escape-последовательности. Пользователям веб это обычно безразлично до того момента, когда потребуется ввести сложный URL в явном виде (броузеры обычно внутренним образом обрабатывают детали escape-кодов). Но если вы пишете сценарии, которые должны генерировать HTML или URL, нужно следить за преобразованием символов, зарезервированных в том или ином синтаксисе.

Поскольку у HTML и URL синтаксис разный, в Python содержатся два разных набора инструментов для преобразования их текста. В стандартной библиотеке Python:

- cgi.escape преобразует текст, который должен быть вставлен в HTML.
- urllib.quote и quote_plus преобразуют текст, который должен быть вставлен в URL.

В модуле urllib есть также средства для обратного преобразования URL (unquote, unquote_plus), но преобразования HTML в целом не обращаются во время синтаксического анализа HTML (htmllib). Чтобы проиллюстрировать действия этих двух соглашений по преобразованию и средствам, применим каждый из этих инструментов к нескольким простым примерам.

Преобразование кода HTML

Как было показано ранее, cgi.escape транслирует код, который должен быть включен в HTML. Обычно эта утилита вызывается в сценариях CGI, но несложно исследовать ее действие и интерактивно:

```
>>> import cgi
>>> cgi.escape('a < b > c & d "spam"', 1)
'a &lt; b &gt; c &amp; d &quot;spam&quot;'

>>> s = cgi.escape("1<2 <b>hello</b>")
```

```
>>> s
'1&lt;2 &lt;b&gt;hello&lt;/b&gt;'
```

Модуль Python `cgi` автоматически преобразует специальные в синтаксисе HTML символы в соответствии с соглашениями HTML. Он транслирует `<`, `>`, `&` и при передаче дополнительного аргумента «истина» также `"` в escape-последовательности вида `&X;`, где `X` является мнемоникой, обозначающей исходный символ. Например, `<` обозначает оператор «меньше» (`<`), а `&` обозначает литеральный амперсанд (`&`).

Средства для обратного преобразования в модуле CGI нет, потому что последовательности escape-кодов HTML распознаются в контексте анализатора HTML, используемом веб-броузером при загрузке страницы. В Python также есть полный анализатор HTML в виде стандартного модуля `htmllib`, импортирующий и уточняющий инструменты модуля `sgmllib` (HTML является разновидностью синтаксиса SGML). Мы не станем здесь вникать в детали средств синтаксического анализа HTML (см. подробности в библиотечном руководстве), но для иллюстрации того, как в конечном итоге escape-коды преобразуются обратно, приведем пример работы модуля SGML, воспроизводящий последнюю строку, выведенную выше:

```
>>> from sgmllib import TestSGMLParser
>>> p = TestSGMLParser(1)
>>> s
'1&lt;2 &lt;b&gt;hello&lt;/b&gt;'
>>> for c in s:
...     p.feed(c)
...
>>> p.close()
data: '1<2 <b>hello</b>'
```

Преобразование URL

В URL в качестве специальных зарезервированы другие символы и используются другие соглашения по преобразованию. По этой причине преобразование URL для передачи осуществляется другими библиотечными средствами Python. Модуль Python `urllib` предоставляет два средства, осуществляющие трансляцию: `quote` для реализации стандартных шестнадцатеричных последовательностей `%XX` escape-кодов URL для большинства не буквенно-цифровых символов и `quote_plus` для дополнительной трансляции пробелов в знаки «плюс». Модуль `urllib` также предоставляет функции для обратного преобразования кодов символов в строке URL: `unquote` преобразует последовательности `%XX`, а `unquote_plus` также преобразует знаки «плюс» обратно в пробелы. Вот пример работы модуля из интерактивного приглашения:

```
>>> import urllib
>>> urllib.quote("a & b #! c")
'a%20%26%20b%20%23%21%20c'

>>> urllib.quote_plus("C:\stuff\spam.txt")
'C%3a%5cstuff%5cspam.txt'

>>> x = urllib.quote_plus("a & b #! c")
>>> x
'a+%26+b+%23%21+c'

>>> urllib.unquote_plus(x)
'a & b #! c'
```

Escape-последовательности URL помещают шестнадцатеричные значения небезопасных символов вслед за знаком `%` (обычно их ASCII-коды). В `urllib` небезопасными символами обычно считаются все, кроме букв, цифр и некоторых безопасных специаль-

ных символов (символов _, . -) и / по умолчанию). Можно индивидуально настроить трансляцию, задав строку безопасных символов в качестве дополнительного аргумента вызова; по умолчанию аргумент имеет значение /, но при передаче пустой строки преобразуется и /:

```
>>> urllib.quote_plus("uploads/index.txt")
'uploads/index.txt'

>>> urllib.quote_plus("uploads/index.txt", '')
'uploads%2findex.txt'
```

Обратите внимание, что модуль Python cgi в процессе извлечения входных данных также транслирует escape-последовательности URL обратно в их исходные символы и меняет знаки + на пробелы. Внутренне cgi.FieldStorage при необходимости автоматически вызывает urllib.unquote для выделения и обратного преобразования параметров, передаваемых в конце URL (трансляция происходит в основном в cgi.parse_qs). В итоге сценарии CGI получают исходные, непреобразованные строки URL, и им не требуется самостоятельно производить обратное преобразование значений. Как мы видели, сценариям CGI даже вообще не требуется знать, что входные данные поступили из URL.

Преобразование URL, встроенных в код HTML

Но что делать с теми URL, которые находятся внутри HTML? То есть как выполнять преобразование, когда генерируется текст, вставляемый в URL, который, в свою очередь, встраивается в генерируемый код HTML? В некоторых предшествующих примерах внутри тегов гиперссылок <A HREF> используются жестко кодированные URL, дополненные входными параметрами; файл *languages2.cgi*, например, выводит HTML, содержащий URL:

```
<a href="getfile.cgi?filename=languages2.cgi">
```

Поскольку URL здесь встроен в HTML, он должен быть, по меньшей мере, преобразован в соответствии с соглашениями HTML (например, все символы < должны быть превращены в <), а все пробелы должны быть преобразованы в знаки +. Этого можно достичь с помощью вызова cgi.escape(url) с последующим string.replace(url, " ", "+"), чего в большинстве случаев должно оказаться достаточно.

Однако в общем случае этого мало, потому что соглашения HTML по escape-последовательностям отличны от соглашений URL. Надежное преобразование URL, встраиваемых в код HTML, следует осуществлять путем применения к строке URL urllib.quote_plus перед тем, как поместить ее в текст HTML. Результат преобразования будет также удовлетворять соглашениям по преобразованию HTML, потому что urllib транслирует больше символов, чем cgi.escape, а % в escape-последовательностях URL не является специальным символом в HTML.

Но здесь есть еще одно препятствие: нужно быть осторожными с символами & в строках URL, которые встраиваются в код HTML (например, в теги гиперссылок <A>). Даже если части строки URL преобразованы согласно кодировке URL, при наличии нескольких параметров они разделяются символом &, а разделитель & также может оказаться преобразованным в & соответственно соглашениям HTML. Чтобы понять почему, рассмотрим такой тег гиперссылки HTML:

```
<A HREF="file.cgi?name=a&job=b&amp=c&sect=d&lt=e">hello</a>
```

При выводе в большинстве проверенных мной броузеров этот URL ссылки выглядит неправильно:

```
file.cgi?name=a&job=b&=c§=d<=e
```

Первые два параметра сохранены, как предполагалось (`name=a`, `job=b`), потому что перед `name` нет `&`, а `&job` не распознается в качестве допустимого в HTML escape-кода символа. Однако `&amp`, `&sect` и `&lt` интерпретируются как специальные символы, потому что служат именами допустимых в HTML escape-кодов. Чтобы это действовало так, как нужно, необходимо преобразовать разделители `&`:

```
<A HREF="file.cgi?name=a&amp;job=b&amp;amp=c&amp;sect=d&amp;lt=e">hello</a>
```

Такую полностью преобразованную ссылку броузеры выводят, как надо:

```
file.cgi?name=a&job=b&amp=c&sect=d&lt=e
```

Из этой истории должен быть сделан следующий вывод: если нет уверенности, что все, кроме самого левого, параметры в строке запроса URL, встраиваемой в HTML, отличны от escape-кодов символов HTML вроде `amp`, в целом нужно пропустить весь URL через `cgi.escape` после того, как имена параметров и их значения будут преобразованы с помощью `urllib.quote_plus`:

```
>>> import cgi
>>> cgi.escape('file.cgi?name=a&job=b&amp=c&sect=d&lt=e')
'file.cgi?name=a&amp;job=b&amp;amp=c&amp;sect=d&amp;lt=e'
```

При этом я должен добавить, что в некоторых примерах этой книги разделители `&` в URL, встраиваемых в HTML, не преобразуются, поскольку известно, что имена параметров в этих URL не конфликтуют с escape-кодами HTML. Однако это не самое общее решение, и при наличии сомнений преобразуйте много и часто.

«Взгляните на жизнь с ее приятной стороны»

Чтобы все эти правила форматирования не показались слишком нескладными (и не заставили вас вскрикивать по ночам!), обратите внимание на то, что эти соглашения по преобразованию HTML и URL диктуются самим Интернетом, а не Python. (Мы видели, что в Python существует другой механизм преобразования специальных символов в строковых константах – с помощью обратных слэшей.) Эти правила проистекают из того обстоятельства, что Сеть основана на идее пересылки по свету форматированных строк, и на них, несомненно, оказала влияние тенденция различных групп по интересам разрабатывать различные системы обозначений.

Можете, однако, утешиться тем, что часто не требуется заботиться о таких таинственных вещах, а если требуется, то Python автоматизирует процесс с помощью библиотечных средств. Просто имейте в виду, что всякому сценарию, генерирующему HTML или URL динамически, возможно, требуется для устойчивости воспользоваться средствами Python преобразования в escape-последовательности. В дальнейших примерах этой и двух следующих глав часто будут использоваться средства преобразования HTML и URL. В главе 15 мы также познакомимся с такими системами, как Zope, стремящимися избавиться от некоторых низкоуровневых сложностей, с которыми сталкиваются разработчики сценариев CGI. И как обычно в программировании, ничто не может заменить интеллект; ценой доступа к поразительным технологиям вроде Интернета является их сложность.

Отправка файлов клиентам и серверам

Пора объяснить часть кода HTML, которую мы держали в тени. Обратили внимание на гиперссылки на главной странице примера селектора языка для показа исходного кода сценария CGI? Обычно мы не видим исходный код такого сценария, потому что обращение к сценарию CGI заставляет его исполняться (мы можем видеть только выдаваемый им HTML, генерируемый для создания новой страницы). Сценарий примера 12.23, на который указывает гиперссылка на главной странице `language.html`, обходит это правило, открывая исходный файл и пересылая его содержимое как часть ответа HTML. Текст помечен тегом `<PRE>` как предварительно форматированный текст и преобразован для передачи в составе HTML с помощью `cgi.escape`.

Пример 12.23. PP2E\Internet\Cgi-Web\Basics\languages-src.cgi

```
#!/usr/bin/python
################################################################
# Вывести код сценария languages.cgi, не запуская его.
################################################################

import cgi
filename = 'languages.cgi'

print "Content-type: text/html\n"       # обернуть в html
print "<TITLE>Languages</TITLE>"
print "<H1>Source code: '%s'</H1>" % filename
print '<HR><PRE>'
print cgi.escape(open(filename).read())
print '</PRE><HR>'
```

Если перейти на этот сценарий в Веб по гиперссылке или введя его URL вручную, то сценарий доставит клиенту ответ, содержащий текст файла с исходным кодом сценария CGI. Он показан на рис. 12.25.

Обратите внимание, что здесь также решающим обстоятельством является форматирование текста файла с помощью `cgi.escape`, потому что он встраивается в код HTML ответа. Если этого не сделать, то все символы текста, имеющие какое-либо значение в коде HTML, будут интерпретированы как теги HTML. Например, символ оператора C++ `<` в тексте этого файла может привести к странным результатам, если не преобразовать его надлежащим образом. Утилита `cgi.escape` превратит его в стандартную последовательность `&lt;`, которая может быть безопасно встроена.

Отображение клиентом произвольных файлов сервера

Почти сразу после того, как я написал сценарий предыдущего примера для просмотра исходного кода languages, мне пришло в голову, что потребует немного дополнительной работы, а принесет много пользы написание обобщенной версии – такой, которая по переданному ей имени файла отобразит *любой* файл на сайте. На стороне сервера это простое видоизменение: необходимо лишь разрешить передачу имени файла в качестве входных данных. Сценарий Python *getfile.cgi* из примера 12.24 реализует это обобщение. Он предполагает, что имя файла введено в форму на веб-странице или дописано в конец URL в качестве параметра. Помните, что модуль Python `cgi` прозрачным образом обрабатывает оба эти случая, поэтому в данном сценарии нет кода, который проводил бы между ними какое-либо различие.

Пример 12.24. PP2E\Internet\Cgi-Web\Basics\getfile.cgi

```
#!/usr/bin/python
```

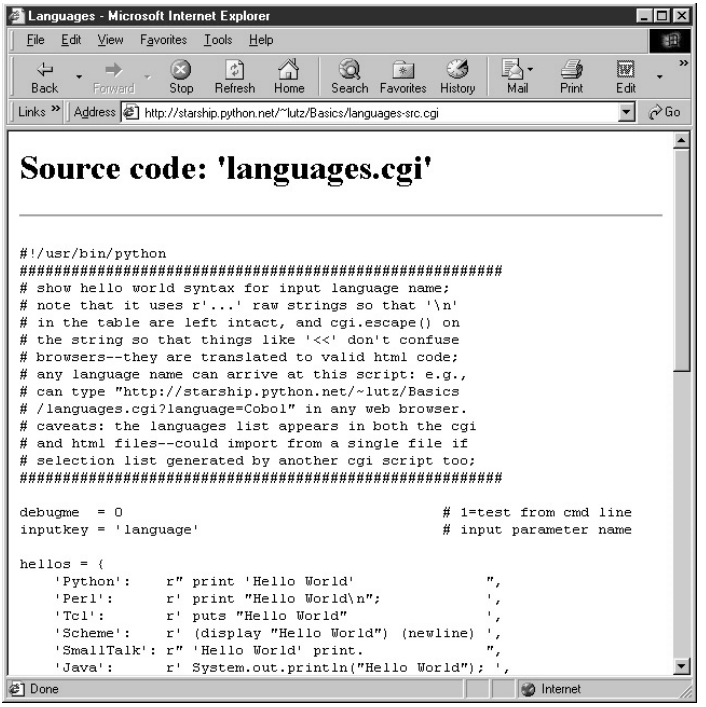


Рис. 12.25. Страница просмотра исходного кода

```
##################################################################################
# Отобразить любой файл cgi (или другой) с сервера без запуска его.
# Имя файла можно передать как параметр URL или поле формы; например,
# http://server/~lutz/Basics/getfile.cgi?filename=somefile.cgi.
# Пользователи могут сохранить файл локально путем "удаления и вставки" или "View source".
# В IE запуск версии text/plain (formatted=0) иногда открывает Notepad, но концы строк
# не всегда имеют формат DOS; Netscape правильно показывает текст на странице броузера.
# Отправка файла в режиме text/html действует в обоих броузерах--текст правильно отображается
# на странице ответа в броузере. Мы также проверяем имя файла, пытаясь не показывать
# закрытые файлы; в целом это может не предотвратить доступа к таким файлам.
##################################################################################

import cgi, os, sys
formatted = 1                                   # 1=обернуть текст в html
privates  = ['../PyMailCgi/secret.py']          # не показывать эти файлы

html = """
<html><title>Getfile response</title>
<h1>Source code for: '%s'</h1>
<hr>
<pre>%s</pre>
<hr></html>"""

def restricted(filename):
    for path in privates:
        if os.path.samefile(path, filename):    # объединить все пути по os.stat
            return 1                            # иначе возвращает None=false
```

```
try:
    form = cgi.FieldStorage()
    filename = form['filename'].value          # параметр url или поле формы
except:
    filename = 'getfile.cgi'                   # или filename по умолчанию

try:
    assert not restricted(filename)            # загрузить, если не закрытый
    filetext = open(filename).read()
except AssertionError:
    filetext = '(File access denied)'
except:
    filetext = '(Error opening file: %s)' % sys.exc_value

if not formatted:
    print "Content-type: text/plain\n"         # отправить простой текст
    print filetext                             # работает в NS, но не в IE
else:
    print "Content-type: text/html\n"          # обернуть в html
    print html % (filename, cgi.escape(filetext))
```

Этот сценарий Python, выполняемый на сервере, просто извлекает имя файла из разобранного объекта входных данных CGI, читает и выводит текст файла для отправки броузеру клиента. В зависимости от значения глобальной переменной `formatted` файл отправляется либо в режиме простого текста (с указанием в заголовке ответа `text/plain`), либо вставленным в определение страницы HTML (`text/html`).

Оба режима (а также другие) в целом работают с большинством броузеров, но Internet Explorer обрабатывает режим простого текста не так элегантно, как это делает Netscape, – при проверке он вывел для просмотра загруженного текста редактор Notepad, но из-за символов конца строки в формате Unix файл был показан как одна длинная строка. (Netscape правильно выводит текст в теле самой веб-страницы ответа.) Режим отображения HTML в современных броузерах работает более переносимым образом. О логике этого сценария в работе с файлами, доступ к которым ограничен, несколько ниже.

Запустим сценарий, набрав его URL в верхней части броузера вместе с именем нужного файла после имени сценария. На рис. 12.26 показана страница, которую мы получим при посещении такого URL:

```
http://starship.python.net/~lutz/Basics/getfile.cgi?filename=languages-src.cgi
```

В теле этой страницы показан текст находящегося на сервере файла, имя которого было передано в конце URL; после получения можно просматривать его текст, выполнять операции удаления и вставки для сохранения в файле клиента и т. д. В действительности теперь, имея такую обобщенную программу просмотра исходного кода, можно заменить гиперссылку на сценарий *languages-src.cgi* в *language.html* на URL такого вида:

```
http://starship.python.net/~lutz/Basics/getfile.cgi?filename=languages.cgi
```

В целях иллюстрации на главной странице HTML в примере 12.16 есть ссылки как на первоначальный сценарий отображения исходного кода, так и на предшествующий URL (без сервера и пути к каталогу, потому что файл HTML и сценарий `getfile` располагаются в одном месте). Действительно такие URL являются непосредственными вызовами (хотя и через веб) нашего сценария Python с явно передаваемыми параметрами имен файлов. Как мы видели, параметры, которые передаются в URL, обрабатываются так же, как данные, веденные в поля формы; для удобства напишем также

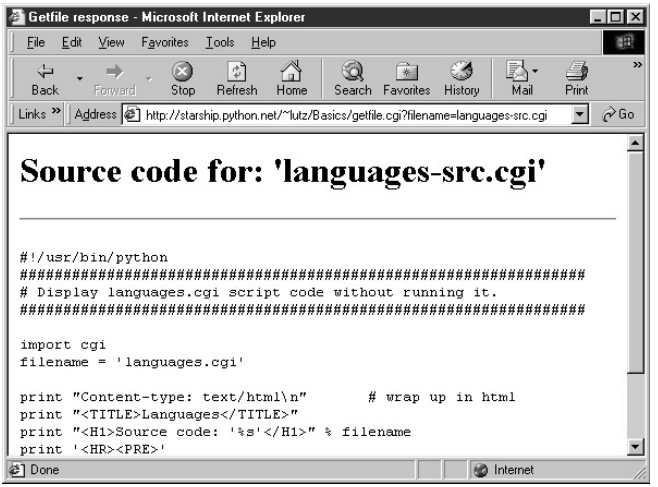


Рис. 12.26. Страница, показываемая общей программой просмотра исходного кода

простую веб-страницу, позволяющую ввести имя нужного файла прямо в форме, как в примере 12.25.

Пример 12.25. PP2E\Internet\Cgi-Web\Basics\getfile.html

```
<html><title>Getfile: download page</title>
<body>
<form method=get action="getfile.cgi">
  <h1>Type name of server file to be viewed</h1>
  <p><input type=text size=50 name=filename>
  <p><input type=submit value=Download>
</form>
<hr><a href="getfile.cgi?filename=getfile.cgi">View script code</a>
</body></html>
```

На рис. 12.27 показана страница, полученная при посещении URL этого файла. На этой странице требуется ввести только имя файла, а не полный адрес сценария CGI.

При нажатии на этой странице кнопки Download для подачи формы имя файла передается на сервер, и мы получаем ту же страницу, что и раньше, когда имя файла до-

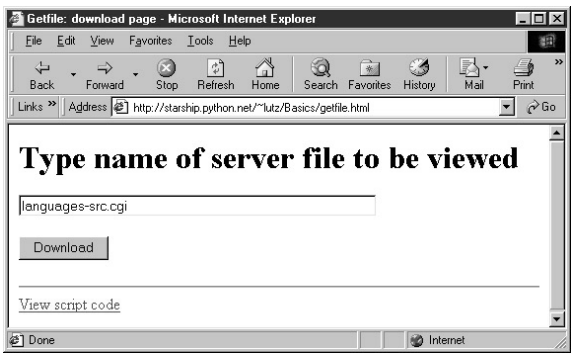


Рис. 12.27. Страница выбора файла для просмотра исходного кода

бавлялось к URL (см. рис. 12.26). На самом деле имя файла здесь тоже будет добавлено к URL: метод `get` в HTML формы указывает броузеру, что он должен присоединить имя файла к URL, точно так же, как мы делали это вручную. Оно появляется в конце URL в поле адреса страницы ответа, хотя мы действительно ввели его в форму.[1]

Обработка закрытых файлов и ошибок

Если у сценария CGI есть права, достаточные для открытия нужного файла на сервере, этот сценарий можно использовать для просмотра и локального сохранения *любого* файла на сервере[2]. Например, рис. 12.28 показывает страницу, полученную от сервера при запросе файла с путем *../PyMailCgi/index.html* (текстовый файл HTML в подкаталоге другого приложения, находящемся в родительском каталоге данного сценария).[3] Пользователи могут указывать как относительные, так и абсолютные пути для доступа к файлу – годится любой синтаксис пути, понятный серверу.

В более общем смысле, этот сценарий отобразит файл с любым путем, для которого у пользователя «nobody» (имя пользователя, под которым обычно выполняются сценарии CGI) есть право на чтение. Это касается почти любого файла на сервере, используемого в веб-приложениях, иначе они прежде всего оказались бы недоступными для броузеров. Способствуя гибкости инструмента, это также представляет потенциальную опасность. Что если мы не хотим, чтобы некоторые файлы сервера могли просматриваться пользователями? Например, в следующей главе будет реализован мо-

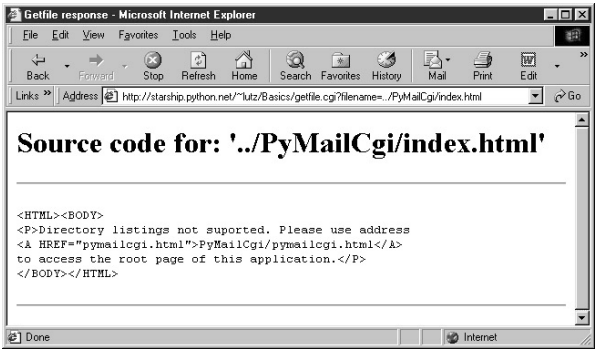


Рис. 12.28. Просмотр файлов с помощью относительных путей

---

[1] Можно заметить одно отличие в ответных страницах, получаемых через форму и через явно введенные URL: для формы значение параметра «filename» в конце URL в ответе может содержать escape-коды URL вместо некоторых символов в пути к файлу, который вы ввели. Броузеры автоматически транслируют некоторые отсутствующие в ASCII символы в escape-коды URL (как `urllib.quote`). Escape-коды URL обсуждались выше в этой главе. Пример такого автоматического преобразования, выполненного броузером, мы увидим чуть ниже.

[2] Стоит заметить, что помещая подобный сценарий на свой веб-сервер, вы можете существенно ослабить защиту этого сервера. Вам будет трудно поместить все файлы, которые вы хотите защитить, в список privates. Полагаю, автор поместил этот пример в книгу исключительно в целях образования, без намерения использовать его на реально работающем сервере. Для того чтобы сделать вышеприведенный сценарий более безопасным, следует, по крайней мере, ограничить область, в которой лежат файлы, доступные для просмотра, одним каталогом и при запросе проверять, находится ли запрашиваемый файл в этом каталоге. – *Примеч. науч. ред.*

[3] PyMailCgi описывается в следующей главе. Если вы ищете исходные файлы для PyErrata (тоже из следующей главы), используйте путь вида *../PyErrata/xxx*. В целом верхний уровень сайта книги соответствует верхнему уровню каталога *Internet/Cgi-Web* в примерах на прилагаемом к книге CD-ROM; `getfile` выполняется в подкаталоге *Basics*.

дуль для шифрования паролей учетных записей электронной почты. Если пользователям будет доступен просмотр исходного кода этого модуля, возможность взлома зашифрованных паролей, передаваемых через Сеть, будет значительно облегчена.

Чтобы уменьшить такую возможность, сценарий `getfile` хранит имена файлов с ограниченным доступом в списке `privates` и с помощью встроенного вызова `os.path.samefile` проверяет, не указывает ли имя запрашиваемого файла на одно из имен в `privates`. Вызов `samefile` сравнивает идентифицирующую информацию для обоих файлов с помощью встроенного вызова `os.stat`, поэтому имена путей, выглядящие синтаксически различными, но ссылающиеся на один и тот же файл, считаются идентичными. Например, на моем сервере следующие пути к модулю шифрования являются разными строками, но вызов `os.path.samefile`[1] для них возвращает «истину»:

```
../PyMailCgi/secret.py
/home/crew/lutz/public_html/PyMailCgi/secret.py
```

Попытка обращения по любому из этих путей влечет вывод страницы с сообщением об ошибке типа показанной на рис. 12.29.

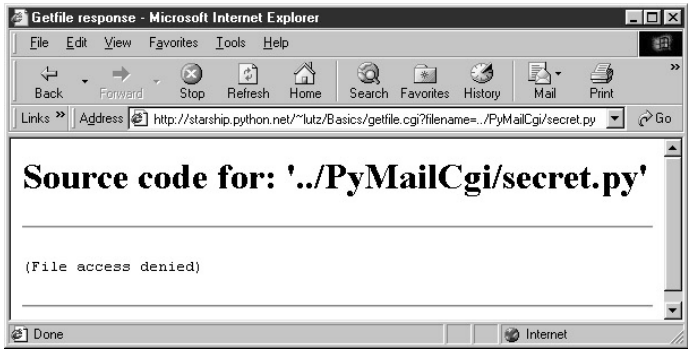


Рис. 12.29. Доступ к закрытым файлам

Заметьте, что настоящие ошибки доступа к файлам обрабатываются иначе. Проблемы прав доступа и доступ к несуществующим файлам, например, перехватываются различными условиями в обработчике исключительных ситуаций и выводят сообщение исключительной ситуации, передавая дополнительный контекст. На рис. 12.30 показана такая страница ошибки.

Общее практическое правило требует подробно сообщать об исключительных ситуациях обработки файлов, особенно во время отладки сценариев. При перехвате таких исключительных ситуаций в сценарии программист должен позаботиться о выводе подробностей (назначение `sys.stderr` в `sys.stdout` не поможет, если Python не выводит сообщение об ошибке). Объекты с типом, данными и стеком трассировки текущей исключительной ситуации всегда доступны в модуле `sys` и могут быть выведены вручную.

---

[1] Вызов `os.path.samefile` работает на машинах POSIX (например, Unix и Linux), но может поддерживаться не всеми машинами, на которых запускается веб-сервер (например, его нет под Windows 98 в Python 1.5.2). При его отсутствии можно поискать другие пути для установления совпадения файлов, например вызвать `os.stat` вручную или проверить совпадение подстрок. Скажем, любая строка пути на сервере, заканчивающаяся после преобразования обратного слэша на *PyMailCgi/secret.py*, скорее всего, указывает на этот закрытый файл.

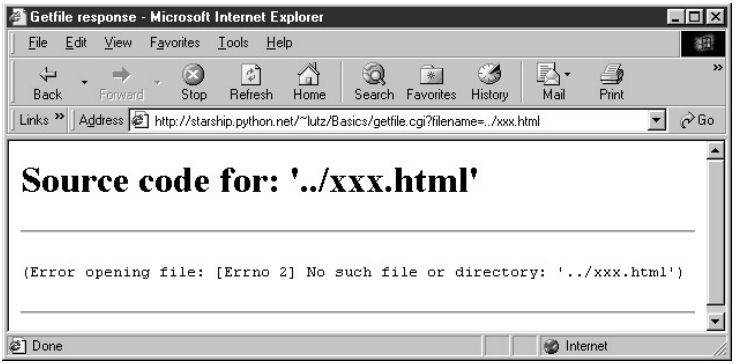


Рис. 12.30. Вывод ошибки обращения к файлу

Проверка по списку закрытых файлов не помешает непосредственно просмотреть модуль шифрования с помощью этого сценария, но нельзя сказать, окажется ли он уязвим для атак со стороны злоумышленников. Безопасность не является темой этой книги, поэтому я не стану развивать ее обсуждение, отметив только, что в Интернете ощутимо присутствует некоторая паранойя. Для систем, устанавливаемых в общедоступном Интернете (в отличие от замкнутых интрасетей), в особенности следует исходить из того, что в конечном счете осуществится худший из возможных сценариев.

Отправка файлов клиента на сервер

Сценарий `getfile` позволяет клиенту просматривать файлы сервера, но в некотором смысле он может считаться инструментом общего назначения для загрузки файлов с сервера. Не так прямо, как при получении файлов по FTP или непосредственно через сокеты, но он служит аналогичным целям. Пользователи сценария могут копировать отображаемый код с веб-страницы или через опцию броузера View Source.

Но как быть с движением в обратном направлении – загрузкой файла с машины клиента на сервер? В предыдущей главе мы видели, что это достаточно просто осуществляется сценарием, выполняемым на стороне клиента и использующим модуль Python для поддержки FTP. Однако такое решение не применимо в контексте веб-броузера; вряд ли вы станете предлагать всем клиентам программы запустить сценарий Python для FTP в другом окне, чтобы отправить файл. Кроме того, нет простого способа явно запросить отправку файла в сценарии на стороне сервера, если только на машине клиента не работает сервер FTP (что случается отнюдь не часто).

Так есть ли способ написать программу для веб, которая разрешает своим пользователям отправлять файлы на общий сервер? На самом деле есть, хотя он больше имеет отношение к HTML, чем собственно к Python. Теги HTML `<input>` поддерживают еще и опцию `type=file`, с помощью которой создается поле ввода вместе с кнопкой, показывающей диалог для выбора файла. Имя файла на машине клиента, который нужно загрузить на сервер, можно ввести в это поле или выбрать с помощью всплывающего диалога. Файл страницы HTML примера 12.26 определяет страницу, которая позволяет выбрать любой файл на стороне клиента и загрузить его в сценарий на стороне сервера, указанный в параметре формы `action`.

Пример 12.26. PP2E\Internet\Cgi-Web\Basics\putfile.html

```
<html><title>Putfile: upload page</title>
<body>
<form enctype="multipart/form-data"
      method=post
      action="putfile.cgi">
  <h1>Select client file to be uploaded</h1>
  <p><input type=file size=50 name=clientfile>
  <p><input type=submit value=Upload>
</form>
<hr><a href="getfile.cgi?filename=putfile.cgi">View script code</a>
</body></html>
```

Нужно обратить внимание на одно ограничение: формы, использующие поля ввода с типом `file`, должны также указывать тип кодировки `multipart/form-data` и метод передачи `post`, как и сделано в этом файле; URL в стиле `get` не годятся для загрузки файлов на сервер. При посещении этого URL показывается страница, приведенная на рис. 12.31. Нажатие кнопки Browse открывает диалог выбора файла, а кнопки Upload – отсылает выбранный файл.

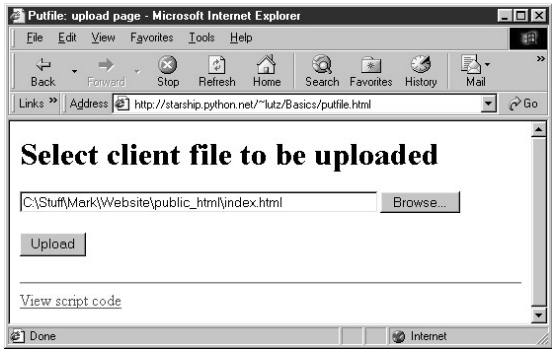


Рис. 12.31. Страница выбора файла для отправки

Когда на стороне клиента нажимается кнопка Upload на этой странице, броузер открывает выбранный файл, читает его и упаковывает его содержимое вместе с остальными полями ввода формы (если они есть). Когда эти данные попадают на сервер, выполняется, как обычно, сценарий Python, указанный в теге формы `action` и приведенный в примере 12.27.

Пример 12.27. PP2E\Internet\Cgi-Web\Basics\putfile.cgi

```
#!/usr/bin/python
###############################################################################
# Извлечь файл, загруженный по http из веб-броузера; пользователи заходят
# на putfile.html, чтобы получить страницу с формой для отправки, которая затем запускает
# этот сценарий на сервере; замечание: способ очень мощный и очень опасный:
# обычно нужно проверять имя файла и т. д. Загрузка возможна, только если есть
# разрешение записи в файл или каталог; команды Unix `chmod 777 uploads`
# может оказаться достаточно; имена путей файлов поступают в формате путей клиента;
###############################################################################
import cgi, string, os, sys
import posixpath, dospath, macpath     # для путей клиента
debugmode    = 0                       # 1=вывод данных формы
```

```
loadtextauto = 0                        # 1=сразу читать файл
uploaddir    = './uploads'              # каталог для записи файлов
sys.stderr = sys.stdout                 # вывод ошибок
form = cgi.FieldStorage()               # разбор данных формы
print "Content-type: text/html\n"       # с пустой строкой
if debugmode: cgi.print_form(form)      # вывод полей формы
# шаблоны html
html = """
<html><title>Putfile response page</title>
<body>
<h1>Putfile response page</h1>
%s
</html>"""

goodhtml = html % """
<p>Your file, '%s', has been saved on the server as '%s'.
<p>An echo of the file's contents received and saved appears below.
</p><hr>
<p><pre>%s</pre>
</p><hr>
"""

# обработать данные формы
def splitpath(origpath):                                # получить файл в конце
    for pathmodule in [posixpath, dospath, macpath]:    # опробовать всех клиентов
        basename = pathmodule.split(origpath)[1]        # сервер может быть любой
        if basename != origpath:
            return basename                             # пропускает пробелы
    return origpath                                     # неудача или нет каталогов

def saveonserver(fileinfo):                             # использовать данные формы с полем file
    basename = splitpath(fileinfo.filename)             # имя без пути каталога
    srvrname = os.path.join(uploaddir, basename)        # записать в каталог, если он задан
    if loadtextauto:
        filetext = fileinfo.value                       # читать текст в строку
        open(srvrname, 'w').write(filetext)             # записать в файл сервера
    else:
        srvrfile = open(srvrname, 'w')                  # иначе читать построчно
        numlines, filetext = 0, ''                      # например, для больших файлов
        while 1:
            line = fileinfo.file.readline()
            if not line: break
            srvrfile.write(line)
            filetext = filetext + line
            numlines = numlines + 1
        filetext = ('[Lines=%d]\n' % numlines) + filetext
    os.chmod(srvrname, 0666)   # разрешить запись, владелец 'nobody'
    return filetext, srvrname

def main():
    if not form.has_key('clientfile'):
        print html % "Error: no file was received"
    elif not form['clientfile'].filename:
        print html % "Error: filename is missing"
    else:
        fileinfo = form['clientfile']
        try:
            filetext, srvrname = saveonserver(fileinfo)
        except:
```

```
            errmsg = '<h2>Error</h2><p>%s<p>%s' % (sys.exc_type, sys.exc_value)
            print html % errmsg
        else:
            print goodhtml % (cgi.escape(fileinfo.filename), cgi.escape(srvrname),
                                                             cgi.escape(filetext))
main()
```

В этом сценарии применены специфические интерфейсы Python для обработки загружаемых на сервер файлов. На самом деле они не очень различаются. Файл доступен сценарию в виде записи в объекте, возвращаемом, как обычно, `cgi.FieldStorage` при анализе формы; ключом для него служит `clientfile`, атрибут `name` поля ввода в коде HTML-страницы.

Однако на этот раз у записи есть дополнительные атрибуты в виде имени этого файла у клиента. Кроме того, при обращении к атрибуту `value` входного объекта загруженного файла содержимое файла автоматически целиком считывается в строку на сервере. Если файл очень большой, его можно читать построчно (или блоками байтов заданного размера). В иллюстративных целях этот сценарий реализует обе схемы: в зависимости от значения глобальной переменной `loadtextauto` он либо запрашивает содержимое файла как строку, либо читает его построчно.[1] Вообще говоря, модуль CGI возвращает для полей отправки файлов объекты со следующими атрибутами:

`filename`

Имя файла, как оно указано у клиента

`file`

Файловый объект, из которого может быть считано содержимое загруженного на сервер файла

`value`

Содержимое загруженного файла (читаемое из файла по требованию)

Существуют дополнительные атрибуты, не используемые в нашем сценарии. Файлы представляют собой третий тип объектов полей ввода; мы уже видели, что атрибут `value` является *строкой* для простых полей ввода, а для элементов с множественным выбором может быть получен *список* объектов.

Для сохранения загруженных файлов на сервере CGI-сценарии (выполняемые пользователем «nobody») должны иметь право записи в каталог, если файл еще не существует, или в сам файл, если он уже есть. Чтобы отдельно разместить загружаемые файлы, сценарий записывает их в тот каталог, который указан в глобальной переменной `uploaddir`. На сервере Linux моего сайта я должен был с помощью `chmod` дать этому каталогу режим 777 (все права чтения/записи/выполнения), чтобы заставить загрузку работать вообще. У вас может быть другой опыт, но если этот сценарий откажется работать, проверьте права доступа.

В этом сценарии также вызывается `os.chmod`, чтобы установить для файла на сервере права доступа, в соответствии с которыми его чтение и запись могут осуществлять все пользователи. Если при загрузке файл создается заново, его владельцем станет «nobody», что означает возможность просмотра файла и загрузки на сервер для всех пользователей в киберпространстве. Однако на моем сервере этот файл будет также по

[1] Обратите внимание, что чтение строк означает, что этот сценарий CGI нацелен на загрузку текстовых, а не двоичных файлов. То, что при этом он также использует режим открытия «w», делает его малопригодным для загрузки двоичных файлов при выполнении на сервере Windows – при записи в данные могут быть вставлены символы `\r`. Если вы забыли, почему это происходит, обратитесь за деталями к главе 2.

умолчанию доступен только для записи пользователю «nobody», что может оказаться неудобным, когда потребуется изменить этот файл вне веб (степень неудобств может различаться в зависимости от операций).

Отделение файлов, загружаемых клиентами, путем помещения их в отдельный каталог на сервере способствует уменьшению угрозы безопасности: нельзя произвольно переписать существующие файлы. Но при этом может потребоваться копирование файлов на сервере после их загрузки, и не устраняются все угрозы безопасности – злонамеренные клиенты сохраняют возможность отправки очень больших файлов, для отслеживания чего нужна дополнительная логика, отсутствующая в данном сценарии. Такие ловушки могут понадобиться только в сценариях, открытых для Интернета в целом.

Если клиент и сервер оба выполнят свои роли, то сценарий CGI, после того как сохранит содержимое файла клиента в новом или существующем уже файле на сервере, предоставит ответную страницу, показанную на рис. 12.32. Для контроля в ответе указываются пути к файлу у клиента и на сервере, а также повтор загруженного файла вместе со счетчиком строк (если чтение производилось в построчном режиме).

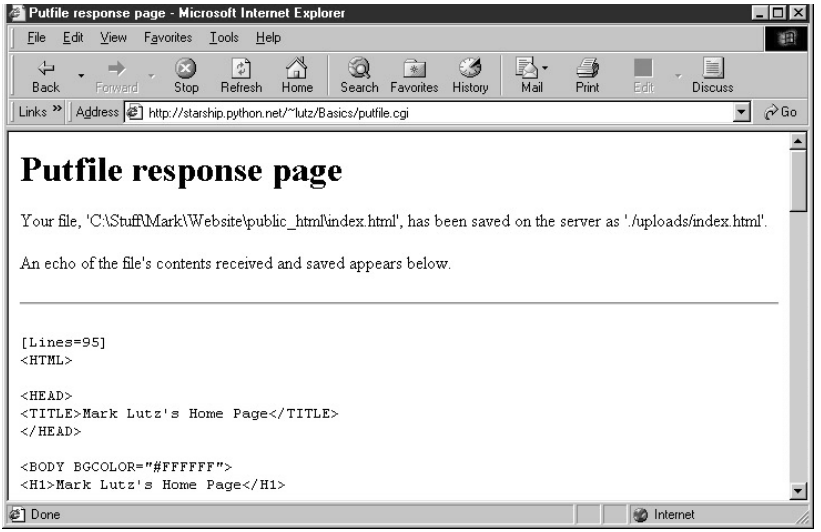


Рис. 12.32. Ответная страница putfile

Кстати, можно проверить загрузку с помощью программы `getfile`, написанной в предыдущем разделе. Просто зайдите на страницу выбора файла и введите путь и имя файла на сервере, как показано на рис. 12.33.

Если предположить, что загрузка файла на сервер произошла успешно, то будет получена результирующая страница программы просмотра, показанная на рис. 12.34. Поскольку пользователь «nobody» (сценариев CGI) смог записать файл, «nobody» должен быть в состоянии и прочесть его.

Обратите внимание на URL в адресном поле этой страницы – броузер перевел символ /, введенный на странице выбора файла, в шестнадцатеричный escape-код `%2F` перед тем, как поместить его в конец URL в качестве параметра. С escape-кодами URL мы познакомились выше в этой главе. В данном случае трансляцию осуществил броузер,

Рис. 12.33. Проверка putfile с помощью getfile – выбор файла

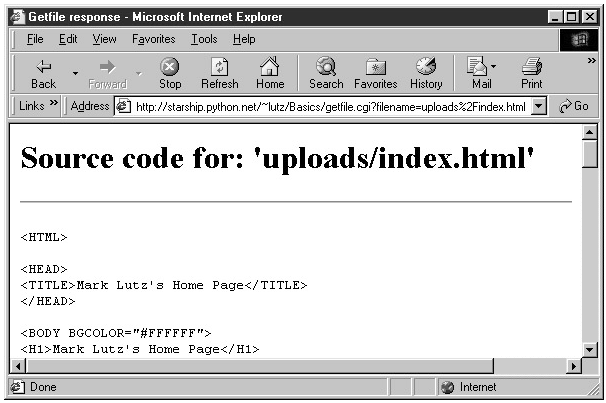


Рис. 12.34. Проверка putfile с помощью getfile – ответ

но конечный результат получился таким же, как если бы мы вручную применили одну из функций преобразования `urllib` к строке пути.

Технически escape-код `%2F` представляет здесь стандартную трансляцию URL для символов, отсутствующих в ASCII, в стандартной схеме кодировки, используемой броузерами. Пробелы обычно также транслируются в символы +.Часто можно обойтись без трансляции большинства отсутствующих в ASCII символов вручную при явной отправке путей (в вводимых URL). Но как мы видели раньше, иногда нужно следить за тем, чтобы преобразовать символы, имеющие в строках URL особое значение (например, `&`), с помощью средств `urllib`.

Учет формата пути, используемого клиентом

В конце своей работы сценарий *putfile.cgi* записывает загруженный файл на сервер в жестко прописанный каталог `uploaddir` под тем именем, которое задано в конце пути к файлу у клиента (то есть без пути каталога на стороне клиента). Обратите, однако, внимание, что функции `splitpath` в этом сценарии приходится делать дополнительную работу для извлечения базового имени файла в правой части. Броузеры посылают имя файла в том формате пути каталога, который используется на машине клиента. Этот формат пути может не совпадать с тем, который используется на сервере, где выполняется сценарий CGI.

Стандартное средство для разделения путей на составные части, `os.path.split`, умеет извлекать базовое имя, но распознает только символы-разделители пути, которые используются на платформе, где она выполняется. Это означает, что если выполнить этот сценарий CGI на машине Unix, `os.path.split` отрубит пути по разделителю /. Однако если пользователь осуществляет отправку файла с машины DOS или Windows, то разделителем в переданном имени файла будет \, а не /. Броузеры, работающие на Macintosh, могут отправлять еще более отличающиеся пути.

Чтобы обрабатывать пути клиентов общим образом, этот сценарий импортирует из библиотеки Python специфические для платформ модули обработки путей для каждого клиента, который должен поддерживаться, и пытается отделить путь поочередно с помощью каждого из них, пока не будет обнаружено имя файла в правой части. Например, `posixpath` обрабатывает пути, отправляемые с платформ в стиле Unix, а `dospath` распознает пути клиентов DOS и Windows. Обычно мы не импортируем эти модули напрямую, поскольку `os.path.split` автоматически загружается с тем из них, который соответствует платформе, на которой работает сервер; однако данный случай – особый, поскольку путь поступает с другой машины. Заметьте, что можно было бы запрограммировать логику разделения пути иным образом, чтобы избежать нескольких вызовов split:

```
def splitpath(origpath):                    # получить имя в конце
    basename = os.path.split(origpath)[1] # попробовать пути сервера
    if basename == origpath:                # без изменений?
        if '\\' in origpath:
            basename = string.split(origpath, '\\')[-1]   # опробовать клиента dos
        elif '/' in origpath:
            basename = string.split(origpath, '/')[-1]    # опробовать клиента Unix
    return basename
```

Но эта альтернативная версия может отказать для некоторых форматов путей (например, для пути DOS с диском, но без обратных слэшей). В настоящем виде в обоих вариантах попусту тратится время, если имя файла уже является базовым (то есть не содержит слева пути к каталогу), но в целом мы должны учитывать более сложные случаи.

Этот сценарий загрузки на сервер работает так, как задумано, но нужно подчеркнуть некоторые замечания, прежде чем закрыть книгу на этом примере:

- Во-первых, `putfile` никак не учитывает несовместимости, существующие между разными платформами в самих именах файлов. Например, пробелы в имени файла, передаваемом клиентом DOS, не транслируются в другие символы; они остаются пробелами в имени файла на стороне сервера, что может быть допустимо, но в некоторых ситуациях затрудняет обработку.
- Во-вторых, этот сценарий тоже ориентирован на загрузку текстовых файлов; он открывает выходной файл в текстовом режиме (при котором коды маркеров конца строки в файле преобразуются в соответствии с символами конца строки, принятыми на машине веб-сервера) и читает входные данные построчно (что может быть неверным для двоичных данных).

Столкнувшись с одним из таких ограничений, вы перейдете в область предлагаемых упражнений.

Способов протолкнуть байты через Сеть много

Наконец, обсудим некоторый контекст. К этому моменту мы уже увидели три сценария `getfile`. Тот, который приведен в этой главе, отличается от двух других, написанных в более ранних главах, но решает сходную задачу:

- В этой главе `getfile` является сценарием CGI на стороне сервера, выводящим файлы через протокол HTTP (на порту 80).
- В главе 10 «Сетевые сценарии» мы создали клиент и сервер `getfile` для передачи файлов через простые сокеты (на порту 50001), а в главе 11 реализовали `getfile` на стороне клиента для передачи через FTP (на порту 21).

Нынешний сценарий `putfile`, основанный на CGI и HTTP, также отличается от основанного на FTP `putfile` из предыдущей главы, но может рассматриваться как альтернатива обеим отправкам, через сокеты и FTP. Чтобы еще раз подчеркнуть эти различия, рис. 12.35 и 12.36 показывают, как новый `putfile` загружает на сервер первоначальный `getfile`, основанный на сокетах.[1]

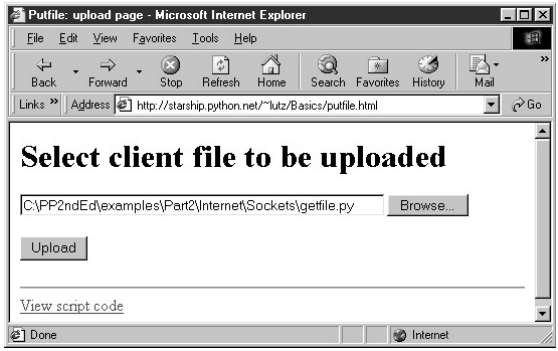


Рис. 12.35. Новый putfile при загрузке getfile, основанного на сокетах

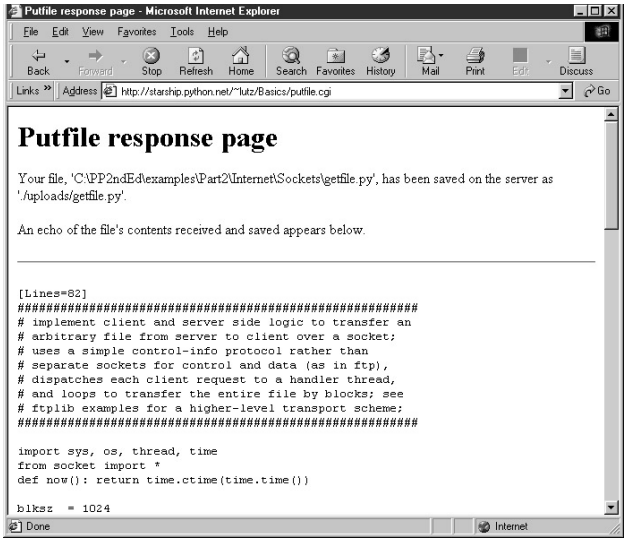


Рис. 12.36. Новый putfile с загруженным getfile, основанным на сокетах

[1] Здесь показана загрузка из более не существующего каталога *Part2* – замените *Part2* на *PP2E*, чтобы указать его настоящее местонахождение, и не удивляйтесь, обнаруживая некоторые отличия в содержимом передаваемых файлов при самостоятельном запуске таких примеров. Как я уже говорил, инженеры обожают вносить изменения.

В действительности CGI-сценарий `getfile` в этой главе лишь отображает файлы, но может считаться средством загрузки, если дополнить его операциями копирования и вставки в веб-броузере. Рис. 12.37 и 12.38 демонстрируют, как `getfile` CGI показывает `getfile`, основанный на сокетах.

Подчеркнуть здесь надо то, что есть много способов передачи файлов через Интернет: сокеты, FTP и HTTP (веб-страницы) – все они могут перемещать файлы между компьютерами. Технически говоря, можно пересылать файлы и с помощью других технологий и протоколов – электронной почты POP, телеконференций NNTP и т. д.

У каждой технологии есть свои особенности, но в итоге они делают одно дело: перемещают биты через Сеть. Все они в конечном счете выполняются через сокеты на определенном порту, но такие протоколы, как FTP, создают дополнительную структуру в слое сокетов, а модели приложений типа CGI создают как структуру, так и возможность программирования.

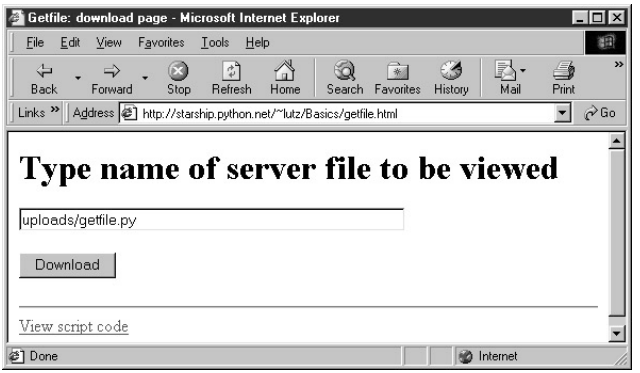


Рис. 12.37. Новый getfile с основанным на сокетах getfile

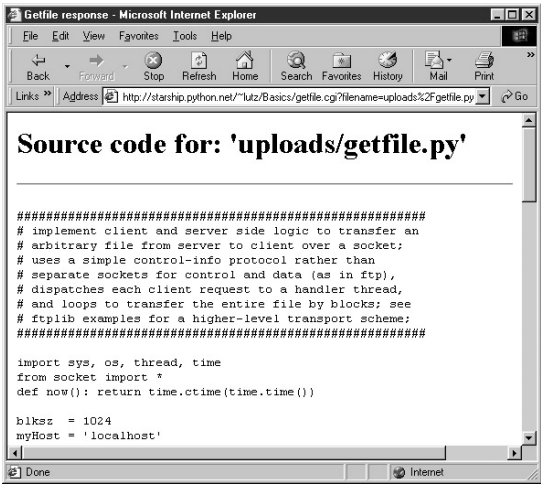


Рис. 12.38. Новый getfile с основанным на сокетах getfile после загрузки

13

Более крупные примеры сайтов, часть 1

«Список дел в Чикаго»

Эта глава представляет собой четвертую часть нашего рассмотрения интернет-программирования на Python и продолжает обсуждение, начатое в предыдущей главе. Там мы изучили основы создания CGI-сценариев на Python для серверов. Вооружившись полученными знаниями, в этой и следующей за ней главах мы перейдем к двум большим конкретным примерам, в которых сделан акцент на более сложные темы CGI:

PyMailCgi

> В данной главе представлен *PyMailCgi*, сайт для чтения и отправки электронной почты, иллюстрирующий концепции системы безопасности, скрытые поля форм, генерацию URL и другие вопросы. Поскольку эта система близка по духу программе PyMailGui, показанной в главе 11 «Сценарии на стороне клиента», этот пример позволяет также сравнить приложения для веб с независимыми от Сети приложениями.

PyErrata

> Глава 14 «Более крупные примеры сайтов, часть 2» представляет *PyErrata,* сайт для сообщения комментариев к книге и обнаруженных ошибок, знакомящий с идеями применения баз данных в области CGI. Эта система демонстрирует стандартные способы постоянного хранения данных на сервере между операциями в веб и способы решения возникающих в модели CGI проблем одновременного обновления данных.

Оба этих конкретных примера основываются на сценариях CGI, но реализуют развитые сайты, выполняющие более практические функции, чем примеры из последней главы.

Как обычно, внимание в этих главах разделено между деталями уровня приложения и принципами программирования на Python. Поскольку оба представленных конкретных примера достаточно велики, они демонстрируют концепции проектирования систем, важные для реальных проектов. Из них также можно больше узнать о сценариях CGI в целом. PyMailCgi, например, знакомит с понятиями сохранения состояния в скрытых полях и URL, а также соображениями безопасности и шифрованием. PyErrata дает возможность изучить концепции постоянных баз данных в контексте сайтов.

Обе системы не отличаются особенной броскостью или богатством функций, встречающимся на сайтах (и правда, первоначальный набросок PyMailCgi был сделан во время задержки в аэропорту Чикаго). Увы, вы не найдете на них ни танцующих медвежат, ни мерцающих огней. С другой стороны, они были написаны для практического использования, более широкого освещения сценариев CGI и для того, чтобы показать, чего можно достичь с помощью программ Python, выполняемых на сервере. В главе 15

«Более сложные темы Интернета» мы изучим системы и инструменты более высокого уровня, основанные на применяемых здесь идеях. А пока порезвимся немного с Python в веб.

Веб-сайт PyMailCgi

В конце главы 11 мы создали программу под названием PyMailGui, в которой с помощью Python+Tk был реализован GUI полного почтового клиента (если вы не читали соответствующий раздел, сейчас может потребоваться бегло взглянуть на него). Теперь мы собираемся сделать нечто в том же роде, но в веб: представленная в данном разделе система PyMailCgi является совокупностью сценариев CGI, реализующих простой основанный на веб интерфейс для отправки и чтения электронной почты в любом броузере.

При изучении этой системы наша задача частично состоит в том, чтобы научиться еще нескольким приемам CGI, частично в том, чтобы получить некоторые сведения о проектировании крупных систем Python в целом, и частично в том, чтобы подчеркнуть соотношение преимуществ и недостатков между системами, реализованными для веб (PyMailCgi), и системами, разработанными для локального выполнения (PyMailGui). Некоторые из этих компромиссов попутно отмечаются в этой главе, а после знакомства с системой мы вернемся к ним и изучим более глубоко.

Общий обзор реализации

На самом верхнем уровне PyMailCgi разрешает пользователям просматривать входящую почту с помощью интерфейса POP и отправлять новую почту по SMTP. Пользователям также предоставлена возможность составить ответ, переадресовать или удалить входящую почту во время ее просмотра. В данной реализации отправить e-mail с сайта PyMailCgi может всякий, но для просмотра почты, как правило, нужно установить PyMailCgi на своем собственном сайте вместе с информацией о своем собственном почтовом сервере (из-за соображений безопасности, излагаемых ниже).

Просмотр и отправка почты выглядят достаточно просто, но в процессе взаимодействия участвует несколько отдельных страниц, для каждой из которых требуется отдельный сценарий CGI или файл HTML. На самом деле PyMailCgi представляет собой довольно линейную систему – в самом сложном варианте схемы взаимодействия с пользователем от начала и до конца имеется шесть состояний (и потому шесть веб-страниц). Так как каждая страница в CGI обычно генерируется отдельным файлом, это предполагает также наличие шести файлов с исходным кодом.

Чтобы помочь уследить за тем, как все файлы PyMailCgi вписываются в общую систему, перед началом практического программирования я написал файл, показанный в примере 13.1. В нем сделан неформальный эскиз прохождения пользователя через систему и вызываемых при этом файлов. Разумеется, для описания передачи управления и информации через состояния можно использовать более формальные обозначения, такие как веб-страницы (например, диаграммы потоков данных), но для данного простого примера достаточно этого файла.

Пример 13.1. PP2E\Internet\Cgi-Web\PyMailCgi\pageflow.txt

```
файл или сценарий                    создает
-----------------                    -------

[pymailcgi.html]                     Корневое окно
 => [onRootViewLink.cgi]             Всплывающее окно пароля
```

```
    => [onViewPswdSubmit.cgi]          Окно списка (загружает всю почту pop)
        => [onViewListLink.cgi]        Окно просмотра + выбор=del|reply|fwd (загрузка)
            => [onViewSubmit.cgi]      Окно редактирования или удалить+подтвердить(удаление)
                => [onSendSubmit.cgi]  Подтверждение (отправляет почту smtp)
                    => назад в корень

=> [onRootSendLink.cgi]                Окно редактирования
    => [onSendSubmit.cgi]              Подтверждение (отправляет почту smtp)
        => назад в корень
```

Этот файл просто перечисляет все файлы с исходным кодом в системе, обозначая запускаемые ими сценарии с помощью символов => и отступа.

Например, ссылки на корневую страницу *pymailcgi.html* вызывают выполняемые сценарии *onRootViewLink.cgi* и *onRootSendLink.cgi*. Сценарий *onRootViewLink.cgi* создает страницу для пароля, кнопка Submit на которой в свою очередь запускает *onViewPswdSubmit.cgi* и т. д. Обратите внимание, что оба действия – просмотр и отправка – могут заканчиваться *onSendSubmit.cgi* для отправки новой почты; операции просмотра попадают в этот сценарий, если пользователь решит ответить на входящую почту или переадресовать ее.

В такой системе сценарии CGI, взятые в отдельности, имеют мало смысла, поэтому полезно помнить общую диаграмму страниц; обращайтесь к этому файлу, если заблудитесь. Дополнительно обстановку показывает рис. 13.1, на котором представлено общее содержание этого сайта при просмотре под Windows функцией PyEdit «Open».

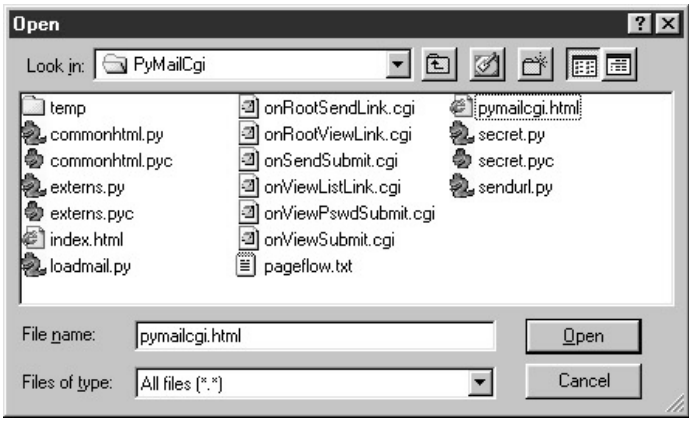


Рис. 13.1. Содержание PyMailCgi

Каталог *temp* используется только во время разработки. Чтобы установить этот сайт, все файлы, которые вы здесь видите, загружаются в подкаталог *PyMailCgi* моего веб-каталога *public_html*. Помимо страниц HTML и файлов сценариев CGI, вызываемых действиями пользователя, PyMailCgi использует также ряд вспомогательных модулей:

- *commonhtml.py* служит библиотекой инструментов HTML.
- *externs.py* объединяет в себе доступ к модулям, импортируемым из других систем.
- *loadmail.py* инкапсулирует загрузку из почтового ящика.
- *secret.py* реализует шифрование пароля с возможностью настройки.

PyMailCgi также повторно использует части модулей *pymail.py* и *mailconfig.py*, созданных в главе 11; на моем веб-сервере они установлены в особый каталог, не обяза-

тельно совпадающий с их местонахождением в дистрибутиве примеров (они находятся в другом каталоге, не показанном на рис. 13.1). Как обычно, PyMailCgi использует также ряд стандартных модулей библиотеки Python: `smtplib`, `poplib`, `rfc822`, `cgi`, `urllib`, `time`, `rotor` и т. п.

Программное обеспечение «ручная кладь»

PyMailCgi действует согласно замыслу и иллюстрирует новые принципы CGI и e-mail, но я хочу заранее сделать несколько предостережений. Приложение было первоначально написано во время двухчасовой задержки в чикагском аэропорту O'Hare (правда, отладка потребовала еще нескольких часов). Я написал его в связи со специфической потребностью – иметь возможность читать и отправлять почту со своего веб-броузера во время путешествий по свету для преподавания Python. Я не стремился к тому, чтобы он доставлял кому-то эстетическое удовольствие и не очень усердствовал над повышением эффективности его выполнения.

Я также умышленно сохранил простоту примера, чтобы включить его в эту книгу. Например, PyMailCgi предоставляет не все функции из имеющихся в программе PyMailGui из главы 11 и перегружает почту чаще, чем, вероятно, следовало бы. Иными словами, следует рассматривать это как систему, работа над которой не завершена; это пока не та программа, которую можно продать. С другой стороны, то, для чего она была предназначена, она делает, и ее можно индивидуально настраивать, внося изменения в ее исходный код Python, чего о всякой продаваемой программе сказать нельзя.

Обзор представления программы

Представление PyMailCgi в такой книге, как эта, составляет проблему, поскольку большинство «действия» сосредоточено в совместно используемых вспомогательных модулях (особенно в одном, под именем *commonhtml.py*); сценарии CGI, которые реализуют взаимодействие с пользователем, сами по себе выполняют немного. Такая архитектура была выбрана намеренно, чтобы сделать сценарии простыми и придать им одинаковый внешний вид. Но это означает, что приходится прыгать между файлами, чтобы понять, как работает система.

Для облегчения усвоения этого примера рассмотрим его код по частям: сначала сценарии страниц, а затем – вспомогательные модули. Во-первых, мы изучим снимки экранов основных веб-страниц, выдаваемых системой, а также файлы HTML и CGI-сценарии Python верхнего уровня, которые их генерируют. Мы начнем с того, что проследим действия при отправке почты, а затем и то, как обрабатывается имеющаяся почта. В этих разделах мы познакомимся с большинством деталей реализации, но для понимания того, что в действительности делают сценарии, необходимо заглянуть вперед в описание вспомогательных модулей.

Я должен также подчеркнуть, что это довольно сложная система, и я не стану описывать ее исчерпывающим образом; читайте попутно исходный код, чтобы разобраться в деталях, о которых явно не сказано в тексте. Весь исходный код системы приведен в этом разделе (и на CD, прилагаемом к книге), и мы изучим здесь ключевые концепции этой системы. Но как обычно в отношении конкретных примеров, приводимых в книге, предполагается, что читатель на данной стадии изучения в состоянии читать код Python и за дополнительными деталями будет обращаться к исходному коду примера. Синтаксис Python настолько близок к исполняемому псевдокоду, что иногда системы лучше описываются на Python, чем на обычном языке.

Корневая страница

Начнем с реализации главной страницы этого примера. Файл, приведенный в примере 13.2, используется преимущественно для вывода ссылок на страницы функций Send и View. Его код написан в виде статического HTML, поскольку здесь нечего генерировать на лету.

Пример 13.2. PP2E\Internet\Cgi-Web\PyMailCgi\pymailcgi.html

```
<HTML><BODY>
<TITLE>PyMailCgi Main Page</TITLE>
<H1 align=center>PyMailCgi</H1>
<H2 align=center>A POP/SMTP Email Interface</H2>
<P align=center><I>Version 1.0, April 2000</I></P>

<table><tr><td><hr>
<P>
<A href="http://rmi.net/~lutz/about-pp.html">
<IMG src="../PyErrata/ppsmall.gif" align=left
alt="[Book Cover]" border=1 hspace=10></A>
This site implements a simple web-browser interface to POP/SMTP email accounts.  Anyone can
send email with this interface, but for security reasons, you cannot view email unless you
install the scripts with your own email account information, in your own server account
directory. PyMailCgi is implemented as a number of Python-coded CGI scripts that run on
a server machine (not your local computer), and generate HTML to interact with the
client/browser.  See the book <I>Programming Python, 2nd Edition</I> for more details.</P>

<tr><td><hr>
<h2>Actions</h2>
<P><UL>
<LI><a href="onRootViewLink.cgi">View, Reply, Forward, Delete POP mail</a>
<LI><a href="onRootSendLink.cgi">Send a new email message by SMTP</a>
</UL></P>

<tr><td><hr>
<P>Caveats: PyMailCgi 1.0 was initially written during a 2-hour layover at Chicago's O'Hare
airport.  This release is not nearly as fast or complete as PyMailGui (e.g., each click
requires an Internet transaction, there is no save operation, and email is reloaded often).
On the other hand, PyMailCgi runs on any web broswer, whether you have Python (and Tk)
installed on your machine or not.

<P>Also note that if you use these scripts to read your own email, PyMailCgi does not guarantee
security for your account password, so be careful out there. See the notes in the View action
page as well as the book for more information on security policies.  Also see:

<UL>
<li>The <I>PyMailGui</I> program in the Email directory, which
        implements a client-side Python+Tk email GUI
<li>The <I>pymail.py</I> program in the Email directory, which
        provides a simple command-line email interface
<li>The Python imaplib module which supports the IMAP email protocol instead of POP
<li>The upcoming openSSL support for secure transactions in the new Python 1.6 socket module
</UL></P>
</table><hr>

<A href="http://www.python.org">
<IMG SRC="../PyErrata/PythonPoweredSmall.gif" ALIGN=left
ALT="[Python Logo]" border=0 hspace=15></A>
<A href="../PyInternetDemos.html">More examples</A>
</BODY></HTML>
```

Файл *pymailcgi.html* описывает корневую страницу системы и располагается в подкаталоге *PyMailCgi* моего веб-каталога, выделенном для этого приложения (и помогающем хранить его файлы отдельно от других приложений). Для доступа к этой системе откройте в броузере адрес:

http://starship.python.net/~lutz/PyMailCgi/pymailcgi.html

Если сделать это, сервер вернет страницу, типа показанной на рис. 13.2.

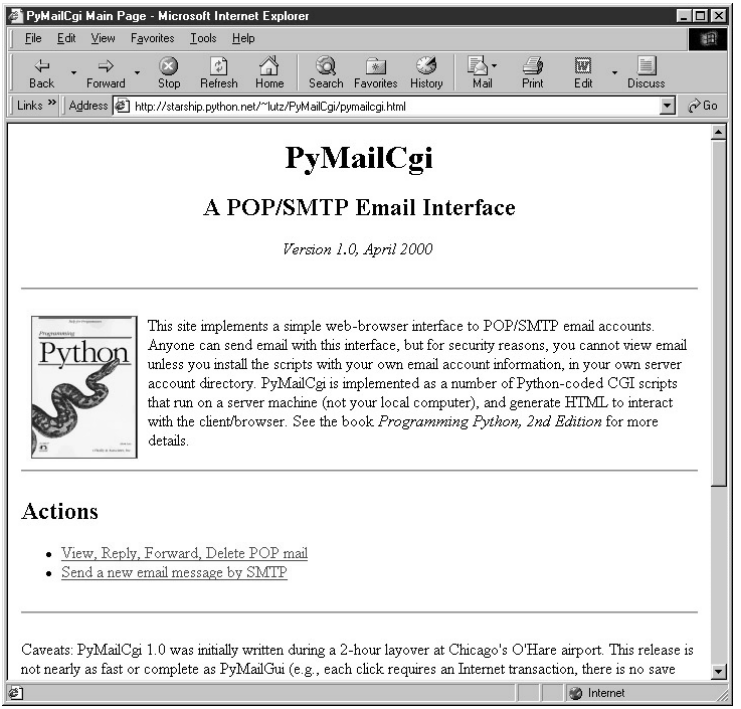


Рис. 13.2. Главная страница PyMailCgi

Теперь, прежде чем вы щелкнете здесь по ссылке View, рассчитывая прочесть свою почту, я должен отметить, что по умолчанию PyMailCgi позволяет любому отправить с этой страницы почту через ссылку Send (как мы уже знаем, в SMTP нет паролей). Однако он не разрешает произвольным пользователям веб прочесть свою почту, не вводя явного и небезопасного URL или не выполняя некоторых действий по установке и настройке. Это сделано нарочно и связано, как будет показано далее, с ограничениями безопасности; как мы увидим позднее, система написана так, что никогда не связывает вместе имя пользователя и пароль электронной почты без применения шифрования.

По умолчанию, следовательно, эта страница устроена так, чтобы читать мою (автора) почтовую учетную запись, и требует для этого моего пароля POP. Так как вы вряд ли угадаете мой пароль (и даже если сделаете это, вряд ли моя почта вам поможет), то PyMailCgi не слишком полезен в том виде, в котором он установлен на этом сайте. Чтобы использовать его для чтения собственной почты, вы должны установить исходный код системы на своем сервере и отредактировать файл конфигурации почты, который мы увидим чуть позже. А пока продолжим, воспользовавшись системой в том виде, в каком она установлена на моем сервере, и с моей учетной записью POP; она работает одинаково независимо от того, к чьей учетной записи обращается.

Отправка почты по SMTP

PyMailCgi поддерживает две главные функции (в виде ссылок на корневой странице): составление и отправка новой почты и просмотр входящей почты. Функция просмотра View ведет на страницы, которые позволяют пользователям отвечать на имеющуюся почту, переадресовывать ее и удалять. Поскольку самой простой является функция отправки Send, начнем с ее страниц и сценариев.

Страница составления сообщений

Функция Send проводит пользователей через две страницы: одну для редактирования сообщения и другую для подтверждения отправки. При щелчке по ссылке Send на главной странице сервер выполняет сценарий примера 13.3.

Пример 13.3. PP2E\Internet\Cgi-Web\PyMailCgi\onRootSendLink.cgi

```
#!/usr/bin/python
# При щелчке 'send' в главном корневом окне

import commonhtml
from externs import mailconfig

commonhtml.editpage(kind='Write', headers={'From': mailconfig.myaddress})
```

Нет, этот файл не был урезан: смотреть в этом файле особенно не на что, поскольку все действия инкапсулированы в модулях `commonhtml` и `externs`. Здесь можно лишь сказать, что этот сценарий вызывает нечто с именем `editpage` для создания ответа, передавая ему нечто с именем `myaddress` для заголовка «From:». Так спроектировано – скрыв детали во вспомогательных модулях, мы значительно облегчим чтение и запись таких сценариев верхнего уровня. В этом сценарии нет также входных данных. При выполнении он создает страницу для составления нового сообщения, показанную на рис. 13.3.

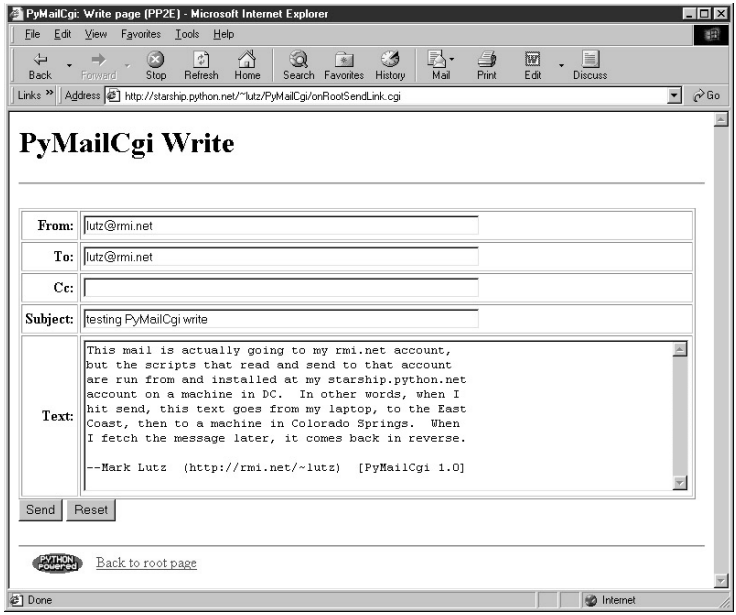


Рис. 13.3. Страница PyMailCgi отправки (составления) сообщения

Сценарий для отправки почты

Эта страница содержит поля для ввода значений стандартных заголовков и самого текста сообщения. Она весьма похожа на клиентскую программу PyMailGui на базе Tkinter, с которой мы познакомились в главе 11. Поле «From:» предварительно заполняется строкой, импортированной из модуля с именем `mailconfig`. Как будет рассказано несколько ниже, в данной системе этот модуль расположен в другом каталоге сервера, но его содержимое то же самое, что в примере PyMailGui. При щелчке по кнопке Send на странице редактирования на сервере выполняется сценарий 13.4.

Пример 13.4. PP2E\Internet\Cgi-Web\PyMailCgi\onSendSubmit.cgi

```
#!/usr/bin/python
# При передаче из окна редактирования - завершить составление,
# ответ или переадресацию

import cgi, smtplib, time, string, commonhtml
#commonhtml.dumpstatepage(0)
form = cgi.FieldStorage()                           # разбор входных данных формы

# имя сервера из модуля или url в стиле get
smtpservername = commonhtml.getstandardsmtpfields(form)

# здесь предполагаются параметры в формате url
from commonhtml import getfield                     # получить атрибуты value
From = getfield(form, 'From')                       # пустые поля могут не посылаться
To   = getfield(form, 'To')
Cc   = getfield(form, 'Cc')
Subj = getfield(form, 'Subject')
text = getfield(form, 'text')

# предупреждение: логика взята из PyMailGui
date  = time.ctime(time.time())
Cchdr = (Cc and 'Cc: %s\n' % Cc) or ''
hdrs  = ('From: %s\nTo: %s\n%sDate: %s\nSubject: %s\n'
                % (From, To, Cchdr, date, Subj))
hdrs  = hdrs + 'X-Mailer: PyMailCgi Version 1.0 (Python)\n\n'

Ccs = (Cc and string.split(Cc, ';')) or []          # некоторые серверы не принимают ['']
Tos = string.split(To, ';') + Ccs                   # cc: строка заголовка и список To
Tos = map(string.strip, Tos)                        # адреса могут содержать ','

try:                                                # smtplib может возбуждать исключение
    server = smtplib.SMTP(smtpservername)           # или возвращать словарь неудачных To
    failed = server.sendmail(From, Tos, hdrs + text)
    server.quit()
except:
    commonhtml.errorpage('Send mail error')
else:
    if failed:
        errInfo = 'Send mail error\nFailed recipients:\n' + str(failed)
        commonhtml.errorpage(errInfo)
    else:
        commonhtml.confirmationpage('Send mail')
```

Этот сценарий получает входную информацию заголовка почты и текста из формы страницы редактирования (или из параметров явного URL) и отправляет сообщение с помощью стандартного модуля Python `smtplib`. Мы подробно изучили `smtplib` в главе 11, поэтому я не стану много говорить о нем сейчас. В действительности код от-

правки почты здесь весьма похож на код PyMailGui (несмотря на то, что я говорил о повторном использовании кода; этот код было бы лучше сделать утилитой).

Утилита `commonhtml` в конечном счете получает имя сервера SMTP, который получит сообщение, из модуля `mailconfig` или входных данных сценария (в поле формы или параметра URL). Если все пройдет успешно, мы получим сгенерированную страницу подтверждения, как на рис. 13.4.

Рис. 13.4. Страница подтверждения отправки PyMailCgi

Обратите внимание, что здесь не видно имени пользователя или пароля: как отмечалось в главе 11, для SMTP нужен только сервер, который слушает на порту SMTP, а не учетная запись пользователя или пароль. В той же главе мы видели, что неудачные операции отправки SMTP либо возбуждают исключительную ситуацию Python (например, если нельзя связаться с хостом сервера), либо возвращают словарь, содержащий получателей, почту которым не удалось отправить.

Если во время доставки почты возникнут проблемы, будет получена страница сообщения об ошибке, типа показанной на рис. 13.5. На этой странице показан получатель, которому почта не была отправлена, – она создана в предложении `else` оператора `try`, в которое мы заключили операцию отправки. Если действительно возникнет исключительная ситуация, выводятся сообщение об ошибке Python и дополнительные данные.

Прежде чем двигаться дальше, скажем, что с помощью этого сценария отправки почты также отсылаются *ответные* и *переадресованные* сообщения для входящей почты POP. Интерфейс для этих операций несколько отличен от используемого для создания нового сообщения с чистого листа, но, как и в PyMailGui, логика обработчика передачи представляет собой тот же самый код – в действительности это просто операции отправки почты.

Стоит также подчеркнуть, что в модуле `commonhtml` инкапсулирована генерация как страниц подтверждения, так и страниц с сообщениями об ошибках, чтобы все такие страницы выглядели в PyMailCgi одинаково независимо от того, где и когда они созданы. Логика, генерирующая страницу редактирования почты в `commonhtml`, повторно используется также в операциях ответа и переадресации (но с другими почтовыми заголовками).

В действительности `commonhtml` заставляет все страницы выглядеть одинаково – он также содержит функции для генерации стандартных заголовков (*header*) и нижних колонтитулов (*footer*), всюду используемых в системе. Вы могли уже заметить, что

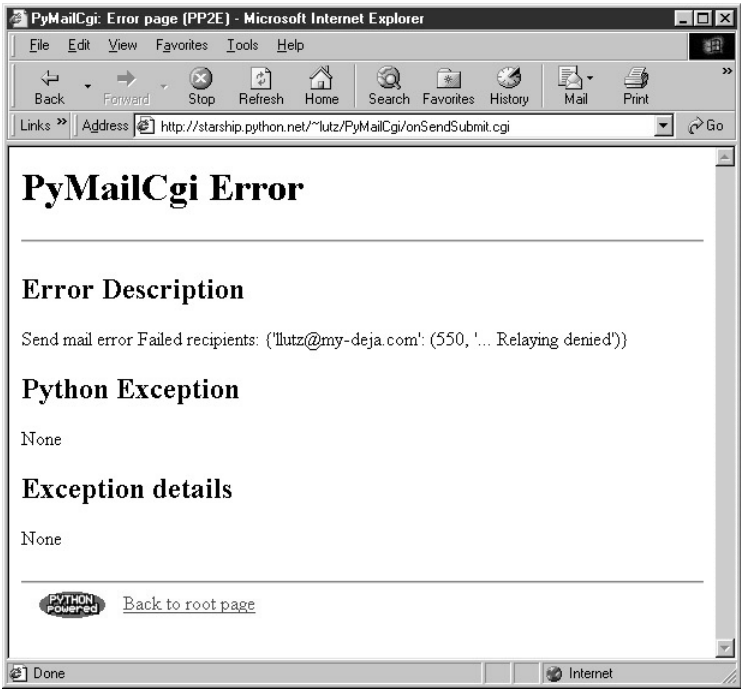


Рис. 13.5. Страница ошибки отправки PyMailCgi

все страницы пока построены по одинаковой схеме: они начинаются с заголовка и горизонтальной черты, содержат что-то свое в середине и заканчиваются внизу другой горизонтальной линией, за которой следуют значок Python и ссылка. Этот стандартный внешний вид создается в `commonhtml`, в котором генерируется все необходимое каждой странице в системе, кроме ее средней части (за исключением корневой страницы – статического файла HTML).

Если вы прямо сейчас хотите узнать, как работает эта инкапсулированная логика, забегите вперед в пример 13.14. Мы рассмотрим его код после того, как изучим остальные страницы этого почтового сайта.

Использование сценария отправки почты без броузера

Вначале я написал сценарий отправки для применения только в PyMailCgi с использованием значений, введенных в форме редактирования почты. Но, как мы видели, входные данные могут отправляться в виде полей формы или параметров URL; так как сценарий отправки почты ищет входные данные в данных CGI, прежде чем импортировать их из модуля `mailconfig`, можно вызывать этот сценарий для отправки почты не только со страницы редактирования. Например, явно введя в броузере URL такого рода (в одну строку без пробелов):

```
http://starship.python.net/~lutz/
     PyMailCgi/onSendSubmit.cgi?site=smtp.rmi.net&
                                From=lutz@rmi.net&
                                To=lutz@rmi.net&
                                Subject=test+url&
                                text=Hello+Mark;this+is+Mark
```

мы действительно отправим почтовое сообщение в соответствии с входными параметрами, указанными в конце. Конечно, такую строку URL долго вводить в поле адреса броузера, но ее можно автоматически генерировать в другом сценарии. Как мы видели в главе 11, с помощью модуля `urllib` можно передать такую строку URL на сервер из программы Python. В примере 13.5 показан один из способов сделать это.

Пример 13.5. PP2E\Internet\Cgi-Web\PyMailCgi\sendurl.py

```
########################################################################
# Отправка почты путем создания из входных данных URL вида:
# http://starship.python.net/~lutz/
#     PyMailCgi/onSendSubmit.cgi?site=smtp.rmi.net&
#                                From=lutz@rmi.net&
#                                To=lutz@rmi.net&
#                                Subject=test+url&
#                                text=Hello+Mark;this+is+Mark
########################################################################

from urllib import quote_plus, urlopen

url = 'http://starship.python.net/~lutz/PyMailCgi/onSendSubmit.cgi'
url = url + '?site=%s'    % quote_plus(raw_input('Site>'))
url = url + '&From=%s'    % quote_plus(raw_input('From>'))
url = url + '&To=%s'      % quote_plus(raw_input('To  >'))
url = url + '&Subject=%s' % quote_plus(raw_input('Subj>'))
url = url + '&text=%s'    % quote_plus(raw_input('text>'))    # или цикл ввода

print 'Reply html:'
print urlopen(url).read()    # html страницы подтверждения или ошибки
```

Запуск этого сценария из командной строки системы дает еще один способ отправить сообщение электронной почты – на этот раз путем обращения к нашему сценарию CGI на удаленном сервере, который должен выполнить всю работу. Сценарий *sendurl.py* выполняется на любой машине, где есть Python и сокеты, позволяет вводить параметры почты интерактивно и вызывает другой сценарий Python, находящийся на удаленной машине. Он выводит HTML, возвращаемый нашим сценарием CGI:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python sendurl.py
Site>smtp.rmi.net
From>lutz@rmi.net
To  >lutz@rmi.net
Subj>test sendurl.py
text>But sir, it's only wafer-thin...
Reply html:
<html><head><title>PyMailCgi: Confirmation page (PP2E)</title></head>
<body bgcolor="#FFFFFF"><h1>PyMailCgi Confirmation</h1><hr>
<h2>Send mail operation was successful</h2>
<p>Press the link below to return to the main page.</p>
</p><hr><a href="http://www.python.org">
<img src="../PyErrata/PythonPoweredSmall.gif"
align=left alt="[Python Logo]" border=0 hspace=15></a>
<a href="pymailcgi.html">Back to root page</a>
</body></html>
```

HTML ответа, выведенный этим сценарием, обычно должен отображаться на новой веб-странице, когда его перехватывает броузер. Такой загадочный вывод нельзя назвать идеальным, но можно легко поискать в нем строку ответа, чтобы определить результат (например, искать «successful» с помощью `string.find`), проанализировать состав с помощью стандартного модуля Python `htmllib` и т. д. Получившееся почтовое

сообщение, которое для разнообразия просмотрим с помощью программы PyMailGui, появляется в моем почтовом ящике и показано на рис. 13.6.

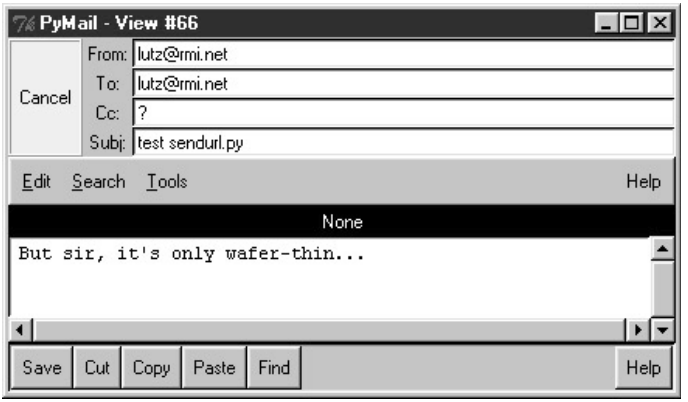


Рис. 13.6. Результат sendurl.py

Конечно, есть другие, менее окольные способы отправки почты с машины клиента. Например, сам модуль Python `smtplib` зависит только от функционирования клиента и соединения с сервером SMTP, в то время как этот сценарий зависит также от машины CGI-сервера (запросы проходят от клиента на сервер CGI и с него на сервер SMTP и обратно). Однако поскольку наш сценарий CGI поддерживает общие URL, он способен на большее, чем тег HTML «mailto:» и может вызываться с помощью `urllib` вне контекста работающего веб-броузера. Например, с помощью сценариев типа *sendurl.py* можно вызывать и *тестировать* программы, выполняемые на сервере.

Чтение почты POP

К настоящему моменту мы проследили тот путь, по которому система *отправляет* новую почту. Теперь посмотрим, что происходит при просмотре входящей почты POP.

Страница ввода пароля POP

Если вернуться на главную страницу, показанную на рис. 13.2, то можно увидеть ссылку View; при нажатии на нее на сервере запускается сценарий, показанный в примере 13.6:

Пример 13.6. PP2E\Internet\Cgi-Web\PyMailCgi\onRootViewLink.cgi

```
#!/usr/bin/python
#########################################################################################
# Выполняется при щелчке по ссылке view на главной странице html; это мог быть файл html,
# поскольку пока в нем нет входных параметров, но я хотел использовать стандартные функции
# заголовков/нижних колонтитулов и показать имена сайта/пользователя для получения почты;
# При передаче формы не посылает имя пользователя вместе с паролем, а передает их только
# как параметры URL или скрытые поля после шифрования пароля с помощью модуля шифрования,
# загружаемого пользователем на сервер; поместить html в commonhtml?
#########################################################################################

# шаблон страницы

pswdhtml = """
<form method=post action=%s/onViewPswdSubmit.cgi>
```

```
<p>
Please enter POP account password below, for user "%s" and site "%s".
<p><input name=pswd type=password>
<input type=submit value="Submit"></form></p>

<hr><p><i>Security note</i>: The password you enter above will be transmitted over the Internet
to the server machine, but is not displayed, is never transmitted in combination with
a username unless it is encrypted, and is never stored anywhere: not on the server (it is only
passed along as hidden fields in subsequent pages), and not on the client (no cookies
are generated). This is still not totally safe; use your browser's back button to back out of
PyMailCgi at any time.</p>
"""

# генерация страницы ввода пароля

import commonhtml                                          # обычный вариант с параметрами:
user, pswd, site = commonhtml.getstandardpopfields({})     # здесь из модуля,
commonhtml.pageheader(kind='POP password input')           # затем из html|url
print pswdhtml % (commonhtml.urlroot, user, site)
commonhtml.pagefooter()
```

Этот сценарий почти целиком состоит из встраиваемого HTML: заключенная в тройные кавычки строка `pswdhtml` выводится с форматированием строки за один шаг. Но поскольку нужно получить имена пользователя и сервера, чтобы показать их на генерируемой странице, использован выполняемый сценарий, а не статический файл HTML. Модуль `commonhtml` загружает имена пользователя и сервера из входных данных сценария (например, если они добавлены в URL сценария) или импортирует их из файла `mailconfig`; в том и другом случае не хочется жестко прописывать их в этом сценарии или его HTML, поэтому файл HTML не подойдет.

Поскольку это сценарий, можно воспользоваться функциями `commonhtml` для создания заголовка и нижнего колонтитула страницы, чтобы генерируемая страница ответа выглядела стандартным образом; она показана на рис. 13.7.

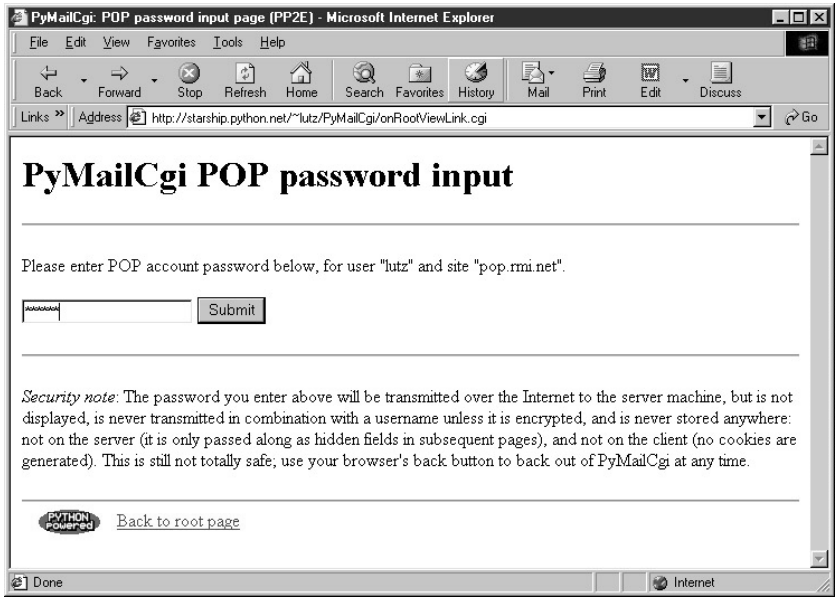


Рис. 13.7. Страница ввода пароля для просмотра почты в PyMailCgi

На этой странице предполагается ввод пользователем пароля учетной записи POP для показанных на ней пользователя и сервера. Обратите внимание, что фактический пароль не показывается; для поля его ввода в HTML определено `type=password`, благодаря чему оно действует, как обычное текстовое поле, но введенные символы отображаются в виде звездочек. (Смотрите также в примере 11.6, как то же сделать в консоли, и в примере 11.23, как то же сделать в GUI.)

Страница выбора почты из списка

После заполнения поля пароля в предыдущей странице и нажатия кнопки Submit пароль передается в сценарий, показанный в примере 13.7.

Пример 13.7. PP2E\Internet\Cgi-Web\PyMailCgi\onViewPswdSubmit.cgi

```
#!/usr/bin/python
# При нажатии submit в окне ввода пароля pop – создать список просмотра

import cgi, StringIO, rfc822, string
import loadmail, commonhtml
from secret import encode      # определяемый пользователем модуль шифрования
MaxHdr = 35                    # максимальная длина заголовков e-mail в списке

# только пароль поступает сюда со страницы, остальное обычно в модуле
formdata = cgi.FieldStorage()
mailuser, mailpswd, mailsite = commonhtml.getstandardpopfields(formdata)

try:
    newmail  = loadmail.loadnewmail(mailsite, mailuser, mailpswd)
    mailnum  = 1
    maillist = []
    for mail in newmail:
        msginfo = []
        hdrs = rfc822.Message(StringIO.StringIO(mail))
        for key in ('Subject', 'From', 'Date'):
            msginfo.append(hdrs.get(key, '?')[:MaxHdr])
        msginfo = string.join(msginfo, ' | ')
        maillist.append((msginfo, commonhtml.urlroot + '/onViewListLink.cgi',
                                {'mnum': mailnum,
                                 'user': mailuser,          # параметры данных
                                 'pswd': encode(mailpswd), # передаются в url,
                                 'site': mailsite}))       # а не в полях
        mailnum = mailnum+1
    commonhtml.listpage(maillist, 'mail selection list')
except:
    commonhtml.errorpage('Error loading mail index')
```

Основным назначением этого сценария является генерация страницы со списком сообщений в почтовом ящике пользователя, используя пароль, введенный на предыдущей странице (или переданный в URL). Как обычно при использовании инкапсуляции большинство деталей скрыто в других файлах:

- `loadmail.loadnewmail` повторно использует почтовый модуль из примера 11.8 для загрузки сообщений по протоколу POP; для вывода списка требуется счетчик сообщений и почтовые заголовки.
- `commonhtml.listpage` генерирует HTML для вывода переданного ему списка кортежей (`text`, `URL`, `parameter-dictionary`) в виде списка гиперссылок на странице ответа; значения параметров находятся в конце URL в ответной странице.

Создаваемый здесь список `maillist` используется при построении тела следующей страницы – списка для выбора сообщений электронной почты, реагирующего на щелчок мышью. Каждая гиперссылка, сгенерированная для этой страницы списка, указывает на сконструированный URL, содержащий достаточно информации, чтобы очередной сценарий мог выбрать и показать конкретное сообщение.

Если все в порядке, этот сценарий сгенерирует HTML-код страницы со списком для выбора почтовых сообщений, как на рис. 13.8. Если вы, как и я, получаете много почты, то, чтобы увидеть конец списка, может понадобиться прокрутить страницу вниз. Она показана на рис. 13.9 и благодаря `commonhtml` соответствует стандартному внешнему виду всех страниц PyMailCgi.[1]

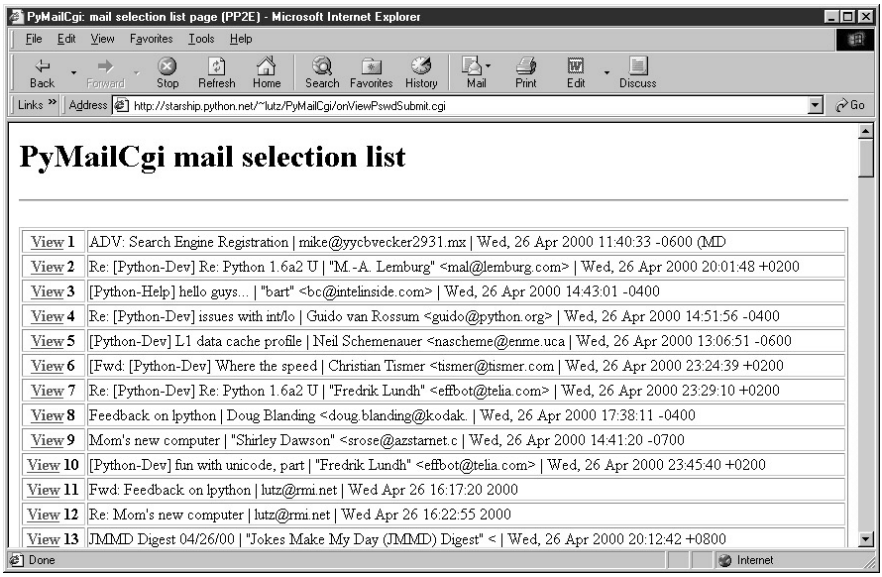


Рис. 13.8. Страница списка выбора сообщений PyMailCgi, верх

Если этот сценарий не может получить доступ к вашей учетной записи почты (например, из-за ввода неправильного пароля), то обработчик в операторе `try` создает страницу сообщения об ошибке в стандартном формате. На рис. 13.10 показана такая страница, сообщающая в ответе об исключительной ситуации Python и ее деталях после перехвата действительной исключительной ситуации.

Передача информации о состоянии в параметрах ссылки URL

Центральным механизмом, действующим в примере 13.7, является генерация таких URL, которые содержат номера сообщений и информацию об учетной записи почты. Щелчок по любой из ссылок View в списке выбора запускает другой сценарий, кото-

[1] Зоркий читатель может заметить, что гиперссылка в нижней части этого экрана показывает пароль POP не в таком формате, как в других местах. Это просто альтернативная схема кодировки из вспомогательного модуля *secret.py*, которая не использует схему кодировки URL, а дает строку чисел ASCII, разделенных черточками. Некоторые читатели могли также обратить внимание на то, что на этой странице в пароле меньше символов, чем в других снимках экранов; поскольку я публикую в этой книге код моего расшифровщика паролей, все пароли здесь часто меняются. А вы действительно думали, что все так просто? :-)

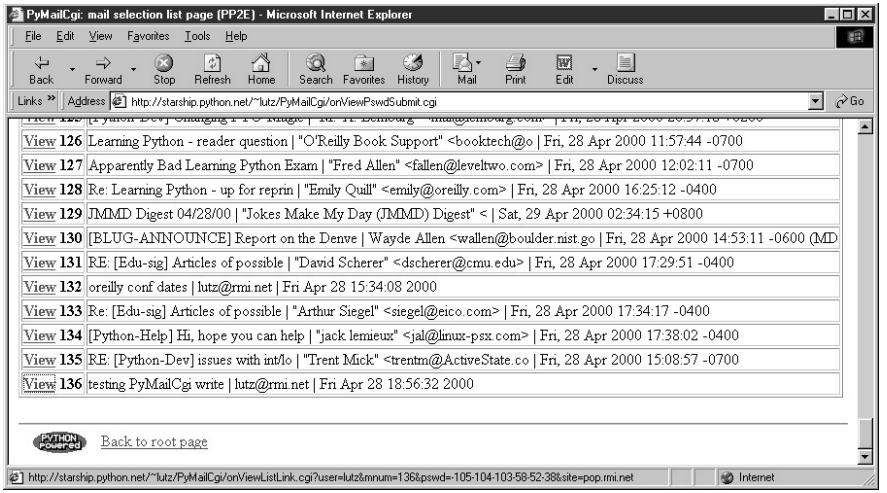


Рис. 13.9. Страница списка выбора сообщений PyMailCgi, низ

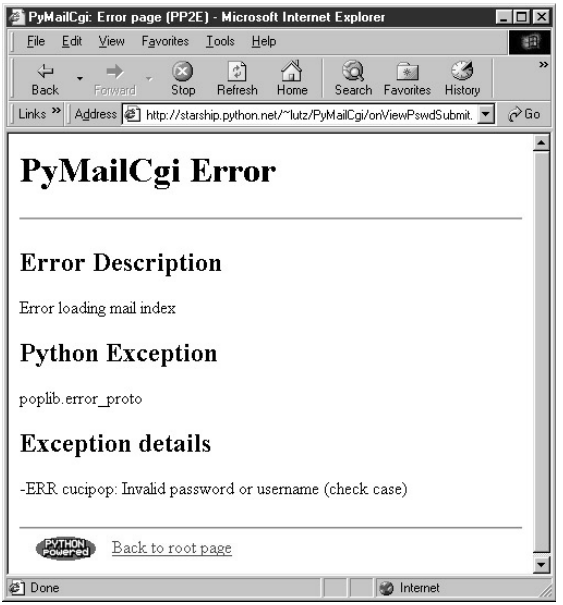


Рис. 13.10. Страница ошибки регистрации PyMailCgi на сервере

рый использует информацию в параметрах URL ссылки для загрузки и вывода выбранного сообщения. Как говорилось в предыдущей главе, поскольку ссылки в списке фактически запрограммированы так, чтобы «уметь» загружать конкретное сообщение, не будет слишком натянутым назвать их умными ссылками (*smart links*) – URL, которые помнят, какое действие должно быть следующим. На рис. 13.11 показана часть HTML, генерируемого этим сценарием.

Все понятно? Если вы не сможете прочесть такой сгенерированный HTML, то сможет ваш броузер. Для читателей, страдающих от ограниченности человеческих возможнос-

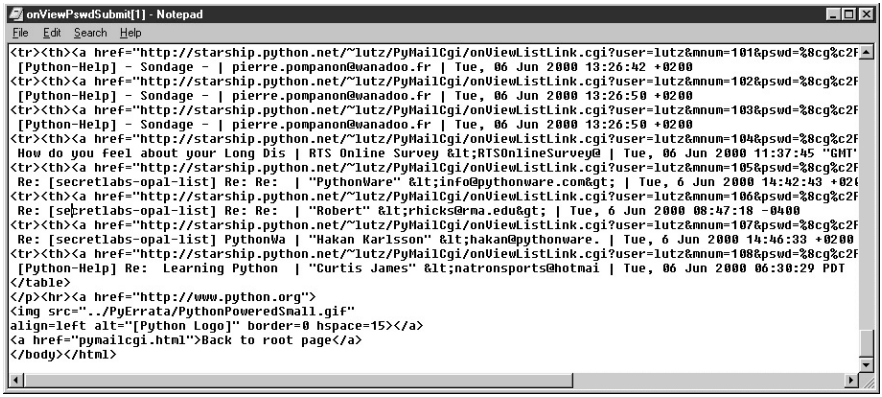


Рис. 13.11. HTML списка просмотра, сгенерированный PyMailCgi

тей синтаксического анализа, приводим отдельно одну из этих ссылок, в которую добавлено форматирование в виде переносов строк и пробелов, облегчающее понимание:

```
<tr><th><a href="http://starship.python.net/~lutz/
                    PyMailCgi/onViewListLink.cgi
                                  ?user=lutz&
                                   mnum=66&
                                   pswd=%8cg%c2P%1e%f3%5b%c5J%1c%f0&
                                   site=pop.rmi.net">View</a> 66
<td>test sendurl.py | lutz@rmi.net | Mon Jun  5 17:51:11 2000
```

PyMailCgi генерирует полностью заданные URL (имена сервера и пути на нем импортируются из общего модуля). Щелчок по слову «View» в гиперссылке, показываемой этим HTML, запускает, как обычно, сценарий `onViewListLink`, которому передаются все параметры, помещенные в конец URL: имя пользователя POP, номер сообщения в POP для письма, связанного с этой ссылкой, а также пароль POP и данные о сайте. Эти значения можно будет получить из объекта, возвращаемого `cgi.FieldStorage` в сценарии, выполняемом следующим. Обратите внимание, что параметр номера сообщения в POP `mnum` различен для каждой ссылки, потому что все они при щелчке открывают разные сообщения, и что текст после `<td>` поступает из заголовков сообщений, извлекаемых с помощью модуля `rfc822`.

Модуль `commonhtml` выполняет кодировку параметров ссылки с помощью модуля `urllib`, а не `cgi.escape`, потому что они являются частью URL. Это очевидно только в параметре пароля `pswd` – его значение зашифровано, но `urllib` дополнительно кодирует небезопасные символы в зашифрованной строке в соответствии с соглашением URL (вот откуда берутся все эти `%xx`). Ничего страшного, если при шифровании получаются странные и даже непечатаемые символы – кодировка URL сделает их доступными для передачи. Когда пароль попадет в следующий сценарий, `cgi.FieldStorage` выполнит обратное преобразование escape-последовательностей URL, убрав из строки с зашифрованным паролем коды `%`.

Полезно посмотреть, как `commonhtml` формирует параметры умных ссылок. Ранее мы узнали, как с помощью `urllib.quote_plus` выполнить кодировку строки перед включением ее в URL:

```
>>> import urllib
>>> urllib.quote_plus("There's bugger all down here on Earth")
'There%27s+bugger+all+down+here+on+Earth'
```

Однако модуль commonhtml вызывает функцию более высокого уровня urllib.urlencode, транслирующую словарь, состоящий из пар *name:value*, в законченную строку параметров URL, которую можно добавлять в URL после маркера ?. Вот пример действия urlencode в интерактивной подсказке:

```
>>> parmdict = {'user': 'Brian',
...             'pswd': '#!/spam',
...             'text': 'Say no more, squire!'}

>>> urllib.urlencode(parmdict)
'pswd=%23%21/spam&user=Brian&text=Say+no+more,+squire%21'

>>> "%s?%s" % ("http://scriptname.cgi", urllib.urlencode(parmdict))
'http://scriptname.cgi?pswd=%23%21/spam&user=Brian&text=Say+no+more,+squire%21'
```

Внутренне urlencode передает каждое имя и значение из словаря во встроенную функцию str (чтобы все они стали строками), а затем пропускает их все через urllib.quote_plus при добавлении к результату. Сценарий CGI строит список аналогичных словарей и передает его в commonhtml для форматирования в виде страницы списка выбора.[1]

Обобщая, такая генерация URL с параметрами предоставляет один из способов передачи информации о состоянии следующему сценарию (наряду с базами данных и скрытыми входными полями форм, обсуждаемыми ниже). Без такой информации о состоянии пользователю пришлось бы заново вводить свое имя, пароль и имя сайта на каждой посещаемой странице. Этим приемом мы воспользуемся вновь в следующем практическом примере, чтобы генерировать ссылки, «умеющие» загружать конкретную запись из базы данных.

Между прочим, генерируемый этим сценарием список не сильно отличается по своим функциям от того, который мы строили в программе PyMailGui в главе 11. Рис. 13.12 показывает представление в этом строго клиентском GUI того же списка сообщений, что и на рис. 13.8 и 13.9.

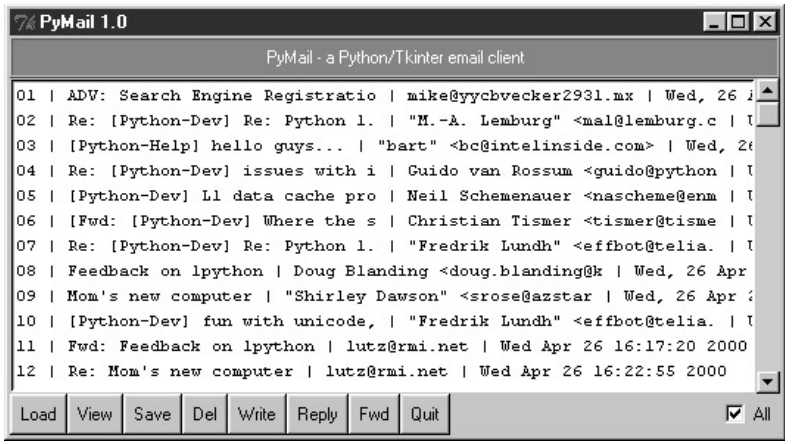


Рис. 13.12. Тот же самый список сообщений в PyMailGui

[1] Формально, кроме того, обычно нужно преобразовывать разделители & в созданных ссылках URL, как при обработке URL с помощью cgi.escape, если имя какого-либо параметра может совпасть с именем escape-кода символа HTML (например, "&high"). Смотрите подробности в предыдущей главе; здесь такое преобразование не производится, потому что нет конфликтов.

Однако PyMailGui использует для создания интерфейса пользователя библиотеку GUI Tkinter, вместо того чтобы посылать HTML в броузер. Он также целиком выполняется на клиенте и по требованию загружает почту с сервера POP на машину клиента через сокеты. Напротив, PyMailCgi выполняется на машине сервера и просто отображает текст почты в броузере клиента – почта загружается с машины сервера POP на сервер *starship*, где выполняются сценарии CGI. У этих архитектурных различий есть некоторые важные разветвления, которые мы обсудим несколько позже.

Протоколы защиты данных

В исходном коде `onViewPswdSubmit` (пример 13.7) обратите внимание на то, что введенный пароль при добавлении в словарь параметров передается в функцию `encode` и потому появляется в URL гиперссылок в зашифрованном виде. Пароли также подвергаются для передачи кодировке URL (с помощью кодов `%`) и позднее раскодируются и расшифровываются в других сценариях при необходимости обратиться к учетной записи POP. Шифрование пароля с помощью `encode` лежит в основе политики безопасности PyMailCgi.

Начиная с Python 1.6 стандартный модуль сокетов должен включать в себя факультативную поддержку *OpenSSL*, open source-реализации защищенных сокетов, для предотвращения перехвата передаваемых в Сети данных. К сожалению, этот пример разрабатывался под Python 1.5.2 и выполняется на сервере, где в Python не было встроенной поддержки защищенных сокетов, поэтому была разработана альтернативная схема, уменьшающая вероятность кражи информации учетной записи почты при перемещении ее по Сети.

Вот как она действует. Когда сценарий вызывается из формы страницы ввода пароля, он получает только один входной параметр: пароль, введенный в форму. Имя пользователя импортируется из модуля `mailconfig`, установленного на сервере, а не передается вместе с незашифрованным паролем (что значительно облегчило бы перехват злоумышленником).

Чтобы передать имя пользователя и пароль следующей странице как информацию о состоянии, этот сценарий помещает их в конец URL страницы списка почтовых сообщений, но только после шифрования пароля с помощью `secret.encode` – функции из модуля, расположенного на сервере, и способного различаться для каждого места, где установлен PyMailCgi. В действительности PyMailCgi написан так, что он не обязан ничего знать о функции шифрования; поскольку шифрование обеспечивается отдельным модулем, можно выбрать тот, который вам больше нравится. Если только вы не опубликуете и свой модуль шифрования, то зашифрованный пароль, передаваемый вместе с именем пользователя, не сильно поможет злоумышленнику.

В результате обычно PyMailCgi не передает и не получает одновременно имя пользователя и пароль в одной операции, если только пароль не зашифрован с помощью выбранной вами функции. Это несколько ограничивает полезность программы (поскольку на сервере можно установить имя пользователя только для одной учетной записи почты), но альтернатива в виде двух страниц – одной для ввода пароля и другой для ввода имени пользователя – была бы еще менее дружелюбной. В целом при желании читать свою почту с помощью этой системы в том виде, как она запрограммирована, необходимо установить ее файлы на своем сервере, поправить *mailconfig.py*, чтобы он отражал детали конкретной учетной записи, и изменить *secret.py* по желанию.

Одно исключение: поскольку любой сценарий CGI можно вызвать с параметрами в явном URL вместо значений полей формы, и так как `commonhtml` пытается получить входные данные из объекта формы, прежде чем импортировать их из `mailconfig`, любой пользователь может проверить свою почту с помощью PyMailCgi, не устанавли-

вая и не настраивая его. Например, URL вида (без переноса строки, добавленного чтобы уместить его на странице):

```
http://starship.python.net/~lutz/PyMailCgi/
 onViewPswdSubmit.cgi?user=lutz&pswd=asif&site=pop.rmi.net
```

действительно загрузит почту в список выбора, какие бы значения имени пользователя, пароля и сайта ни были в него помещены. Из списка выбора можно затем осуществлять просмотр почты, отвечать на нее, переадресовывать и удалять. Обратите внимание, что при взаимодействии в этом месте пароль, отправляемый в URL такого вида, *не* шифруется. В последующих сценариях, однако, предполагается отправка введенного пароля в зашифрованном виде, что затрудняет их использование с явными URL (потребуется соответствие с зашифрованной формой, создаваемой на сервере модулем `secret`). Пароли шифруются при добавлении их в ссылки из списка выбора страницы ответа и остаются зашифрованными в последующих URL и скрытых полях форм.

Пожалуйста, не используйте URL такого типа, если только вам не безразлично, что ваш почтовый пароль окажется открытым. Пересылка одновременно незашифрованных ID пользователя почты и пароля через Сеть в таком URL крайне небезопасна и делает их широко открытыми для любопытных. В действительности это все равно, что дать им заряженное ружье – всякий, перехвативший этот URL, получит полный доступ к вашей учетной записи почты. Еще более ненадежным делает его тот факт, что формат URL опубликован в книге, которая будет широко распространятся по всему свету.

Если вас заботит безопасность и вы хотите пользоваться PyMailCgi, установите его на собственном сервере и настройте `mailconfig` и `secret`. Это должно, по крайней мере, гарантировать, что информация с вашим именем и паролем никогда не будет передаваться незашифрованной в одной транзакции. Эта схема все же не является надежной, поэтому будьте осторожны. Без защищенных сокетов Интернет станет средой типа «используйте на свой страх и риск».

Страница просмотра сообщений

Вернемся к нашей последовательности страниц. В настоящий момент мы все еще рассматриваем список выбора сообщений на рис. 13.8. Если щелкнуть по одной из этих сгенерированных гиперссылок, то умный URL запустит на сервере сценарий примера 13.8, пересылая номер выбранного сообщения и информацию о почтовой учетной записи (пользователь, пароль и сайт) как параметры в конце URL сценария.

Пример 13.8. PP2E\Internet\Cgi-Web\PyMailCgi\onViewListLink.cgi

```
#!/usr/bin/python
###############################################################################################
# Вызывается при щелчке пользователем по ссылке на сообщение в главном списке выбора;
# cgi.FieldStorage делает обратные преобразования escape-кодов во входных
# параметрах ссылки, созданных urllib (%xx и '+' для пробелов уже обращены);
###############################################################################################

import cgi, rfc822, StringIO
import commonhtml, loadmail
from secret import decode
#commonhtml.dumpstatepage(0)
```

```
form = cgi.FieldStorage()
user, pswd, site = commonhtml.getstandardpopfields(form)
try:
    msgnum   = form['mnum'].value                                  # из ссылки url
    newmail  = loadmail.loadnewmail(site, user, decode(pswd))
    textfile = StringIO.StringIO(newmail[int(msgnum) - 1])         # не делайте eval!
    headers  = rfc822.Message(textfile)
    bodytext = textfile.read()
    commonhtml.viewpage(msgnum, headers, bodytext, form)           # зашифрованный пароль
except:
    commonhtml.errorpage('Error loading message')
```

И снова большая часть работы здесь выполняется в модулях `loadmail` и `commonhtml`, которые приведены ниже в этом разделе (примеры 13.12 и 13.14). Этот сценарий вводит дополнительную логику для расшифрования переданного пароля (с помощью конфигурируемого модуля шифрования `secret`) и извлечения заголовков и текста выбранных сообщений с помощью модулей `rfc822` и `StringIO` — так же, как это делалось в главе 11.[1]

Если сообщение успешно загружено и проанализировано, результирующая страница (показанная на рис. 13.13) позволяет просматривать текст сообщения, но не редактировать его. Функция `commonhtml.viewpage` создает параметр HTML «read-only» («только для чтения») для всех текстовых элементов на этой странице.

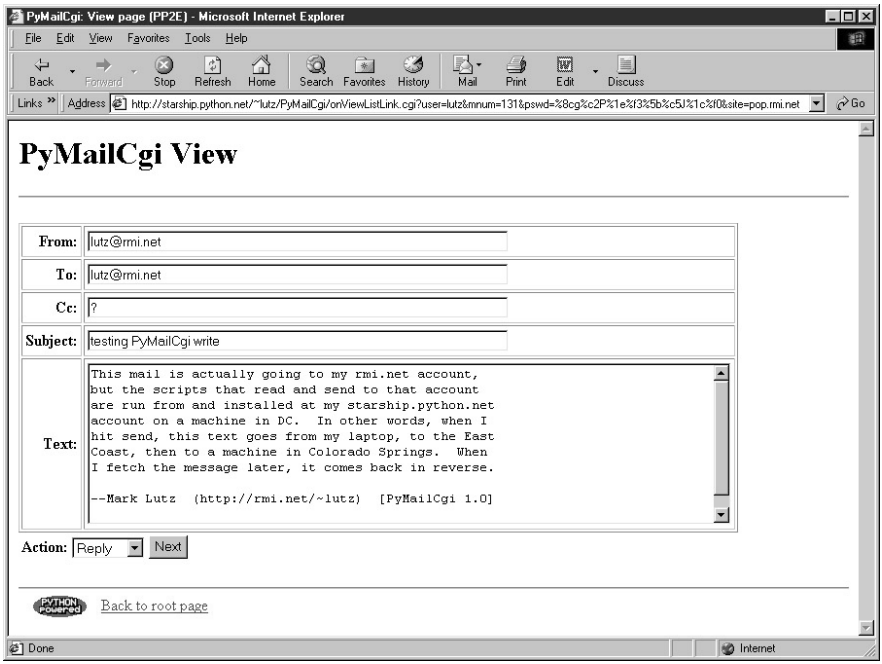


Рис. 13.13. Страница просмотра сообщения PyMailCgi

[1] Обратите внимание, что номер сообщения передается в виде строки и использовать его для загрузки сообщения нужно после преобразования в целое число. Но нельзя выполнять здесь преобразование с помощью `eval`, так как эта строка передана через Сеть и могла быть включена в конец произвольного URL (помните старое предупреждение по этому поводу?).

В нижней части этих страниц просмотра есть выпадающий список выбора действия; если нужно что-то сделать с письмом, выберите в этом списке действие (Reply, Forward или Delete) и щелкните по кнопке Next, чтобы перейти к следующему экрану. Если вы настроились просто побродить по Сети, щелкните внизу по ссылке «Back to root page» и вернетесь на главную страницу либо с помощью кнопки Back броузера вернитесь на страницу списка выбора.

Передача информации о состоянии в скрытых полях ввода HTML

То, чего вы не видите в странице просмотра на рис. 13.13, так же важно, как то, что вы видите. За деталями мы должны отослать вас к примеру 13.14, но здесь происходит нечто новое. Номер исходного сообщения, как и пользователь POP и (по-прежнему зашифрованный) пароль, переданные этому сценарию в составе URL умной ссылки, оказываются скопированными в HTML этой страницы в виде значений скрытых («hidden») полей ввода формы. Код `commonhtml`, который создает скрытые поля, выглядит так:

```
print '<form method=post action="%s/onViewSubmit.cgi">' % urlroot
print '<input type=hidden name=mnum value="%s">' % msgnum
print '<input type=hidden name=user value="%s">' % user # со страницы|url
print '<input type=hidden name=site value="%s">' % site # для удаления
print '<input type=hidden name=pswd value="%s">' % pswd # зашифрованный pswd
```

Подобно параметрам в генерируемых гиперссылках, скрытые поля в HTML-кода страницы позволяют встраивать информацию о состоянии внутрь самой веб-страницы. Увидеть эту скрытую информацию можно только в исходном коде страницы, потому что скрытые поля не отображаются. Но при щелчке по кнопке Submit этой формы скрытые значения автоматически передаются очередному сценарию вместе с теми полями, которые на форме видны.

На рис. 13.14 показан исходный код, сгенерированный для страницы просмотра другого сообщения; скрытые поля ввода, используемые для передачи информации о состоянии выбранного сообщения, помещены в начале файла.

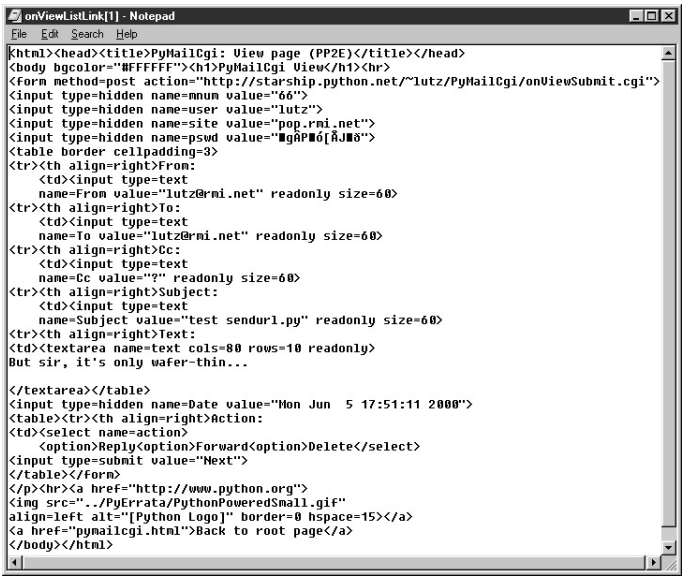


Рис. 13.14. HTML, сгенерированный для страницы просмотра PyMailCgi

В результате скрытые поля ввода в HTML, так же как параметры в конце генерируемых URL, действуют в качестве временного хранилища и сохраняют состояние, передавая его между страницами и этапами взаимодействия с пользователем. Те и другие представляют собой веб-эквивалент переменных в языках программирования. Их удобно использовать, когда приложению требуется что-то передать от одной страницы другой.

Особенно полезными скрытые поля оказываются, когда невозможно вызвать следующий сценарий с помощью гиперссылки со сгенерированным URL, содержащим параметры. Например, следующее действие в нашем сценарии выполняется по кнопке передачи формы (Next), а не по гиперссылке, поэтому для передачи состояния используются скрытые поля. Как и прежде, без этих скрытых полей пользователи должны были бы снова вводить сведения об учетной записи POP в странице просмотра, если они нужны очередному сценарию (в нашем случае они нужны, если следующим действием является Delete).

Кодировка текста сообщения и паролей в HTML

Обратите внимание, что все видимое на странице просмотра сообщения на рис. 13.13 подверглось кодировке с помощью `cgi.escape`. Поля заголовков и сам текст сообщения могут содержать специальные символы HTML и должны транслироваться как обычно. Например, поскольку некоторые почтовые программы позволяют отправлять сообщения в формате HTML, текст сообщения может содержать тег `</textarea>`, который без преобразования безнадежно испортит ответную страницу.

Здесь есть одна тонкость: escape-коды в HTML важны только тогда, когда текст первоначально посылается броузеру (сценарием CGI). Если этот текст затем передается другому сценарию (например, при отправке ответа), текст вернется в свой исходный непреобразованный формат, когда будет снова получен на сервере. Броузер анализирует escape-коды и не возвращает их обратно при отправке данных формы, поэтому в дальнейшем не требуется делать раскодирование. Вот, например, часть области текста после кодировки, посылаемого броузеру во время операции Reply (воспользуйтесь опцией броузера View Source, чтобы посмотреть это в реальности):

```
<tr><th align=right>Text:
<td><textarea name=text cols=80 rows=10 readonly>
more stuff

--Mark Lutz  (http://rmi.net/~lutz)  [PyMailCgi 1.0]

&gt; -----Original Message-----
&gt; From: lutz@rmi.net
&gt; To: lutz@rmi.net
&gt; Date: Tue May  2 18:28:41 2000
&gt;
&gt; &lt;table&gt;&lt;textarea&gt;
&gt; &lt;/textarea&gt;&lt;/table&gt;
&gt; --Mark Lutz  (http://rmi.net/~lutz)  [PyMailCgi 1.0]
&gt;
&gt;
&gt; &gt; -----Original Message-----
```

После отправки этого ответа его текст выглядит так же, как перед преобразованием (и в точности, каким он был у пользователя на странице редактирования сообщения):

```
more stuff

--Mark Lutz  (http://rmi.net/~lutz)  [PyMailCgi 1.0]

> -----Original Message-----
```

```
> From: lutz@rmi.net
> To: lutz@rmi.net
> Date: Tue May  2 18:28:41 2000
>
> <table><textarea>
> </textarea></table>
> --Mark Lutz  (http://rmi.net/~lutz)  [PyMailCgi 1.0]
>
>
> > -----Original Message-----
```

Вы обратили внимание на необычные символы в скрытом поле пароля на снимке экрана, сгенерированного HTML (рис. 13.14)? Оказывается, пароль POP остается зашифрованным при помещении в скрытые поля HTML. Так и должно быть по соображениям безопасности: значения скрытых полей страницы можно увидеть при просмотре исходного кода в броузере, и нельзя исключить возможность перехвата текста этой страницы при передаче в Сети.

Однако пароль уже не преобразован в кодировку URL, когда помещается в скрытое поле, как при нахождении в конце URL умной ссылки. Пароль, сгенерированный здесь как значение скрытого поля, может сейчас содержать непечатаемые символы, что определяется модулем шифрования, но броузеру это безразлично, поскольку поле пропускается через `cgi.escape`, как и все другое, помещаемое в ответный поток HTML. Модуль `commonhtml` заботится о том, чтобы при создании страницы просмотра направить весь текст и заголовки через `cgi.escape`.

Для сравнения на рис. 13.15 показано, как выглядит сообщение, приведенное на рис. 13.13, при просмотре в PyMailGui – использующей Tkinter почтовой утилите клиентской стороны из главы 11. PyMailGui не требуется беспокоиться о таких вещах, как передача состояния в URL или скрытых полях (он хранит состояние в переменных Python) или перекодировка HTML и строк URL (броузеры не используются, а после загрузки почты передачи через сеть не происходит). Что для него требуется, так это наличие Python у клиента, но до этого мы доберемся через несколько страниц.

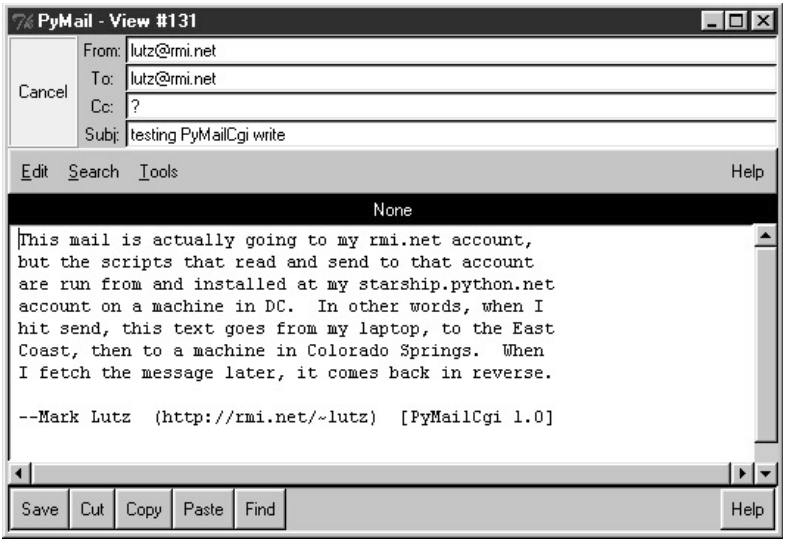


Рис. 13.15. Просмотр того же сообщения в PyMailGui

Страницы действий с сообщением

В данный момент нашего гипотетического взаимодействия в веб с PyMailCgi мы просматриваем почтовое сообщение (рис. 13.13), выбранное на странице со списком. Если, находясь на странице просмотра сообщения, выбрать в выпадающем списке действие и щелкнуть по кнопке Next, то будет вызван сценарий примера 13.9, чтобы выполнить операцию создания ответа, переадресации или удаления для выбранного сообщения.

Пример 13.9. PP2E\Internet\Cgi-WebPyMaiCgi\onViewSubmit.cgi

```
#!/usr/bin/python
# При нажатии кнопки в окне просмотра почты выполняется действие =(fwd, reply, delete)

import cgi, string
import commonhtml, secret
from   externs import pymail, mailconfig
from   commonhtml import getfield

def quotetext(form):
    """
    обратите внимание, что заголовки здесь поступают с формы предыдущей
    страницы, а не от повторного анализа сообщения; это значит,
    что commonhtml.viewpage должен передавать дату как скрытое поле
    """
    quoted = '\n-----Original Message-----\n'
    for hdr in ('From', 'To', 'Date'):
        quoted = quoted + '%s: %s\n' % (hdr, getfield(form, hdr))
    quoted = quoted + '\n' +   getfield(form, 'text')
    quoted = '\n' + string.replace(quoted, '\n', '\n> ')
    return quoted

form = cgi.FieldStorage()  # анализ данных формы или url
user, pswd, site = commonhtml.getstandardpopfields(form)

try:
    if form['action'].value   == 'Reply':
        headers = {'From':    mailconfig.myaddress,
                   'To':      getfield(form, 'From'),
                   'Cc':      mailconfig.myaddress,
                   'Subject': 'Re: ' + getfield(form, 'Subject')}
        commonhtml.editpage('Reply', headers, quotetext(form))

    elif form['action'].value == 'Forward':
        headers = {'From':    mailconfig.myaddress,
                   'To':      '',
                   'Cc':      mailconfig.myaddress,
                   'Subject': 'Fwd: ' + getfield(form, 'Subject')}
        commonhtml.editpage('Forward', headers, quotetext(form))

    elif form['action'].value == 'Delete':
        msgnum = int(form['mnum'].value)       # или string.atoi, но не eval()
        commonhtml.runsilent(                  # здесь нужно поле mnum
            pymail.deletemessages,
                (site, user, secret.decode(pswd), [msgnum], 0) )
        commonhtml.confirmationpage('Delete')

    else:
       assert 0, 'Invalid view action requested'
except:
    commonhtml.errorpage('Cannot process view action')
```

Этот сценарий получает всю информацию о выбранном сообщении как данные входных полей формы (некоторые из которых скрыты) вместе с именем выбранного действия. Следующий шаг при взаимодействии зависит от выбранного действия:

- Действия Reply и Forward генерируют страницу редактирования сообщения, в которой строки исходного сообщения автоматически цитируются с указанием перед каждой символа >.
- Операции удаления вызывают немедленное удаление просматриваемого сообщения с помощью инструмента, импортированного из модуля `pymail` из главы 11.

Во всех этих действиях используются данные, передаваемые из формы предыдущей страницы, но только для действия Delete требуются имя пользователя и пароль POP и раскодирование полученного пароля (он поступает из скрытых полей ввода формы, генерируемых в HTML предыдущей страницы).

Ответ и переадресация

Если в качестве очередного действия выбрать Reply, сценарий генерирует страницу редактирования сообщения, приведенную на рис. 13.16. Текст этой страницы можно редактировать, а при нажатии на ней кнопки Send запускается сценарий отправки почты, который мы видели в примере 13.4. Если все пойдет удачно, будет получена та же страница подтверждения, которую мы получали раньше при написании новой почты с чистого листа (рис. 13.4).

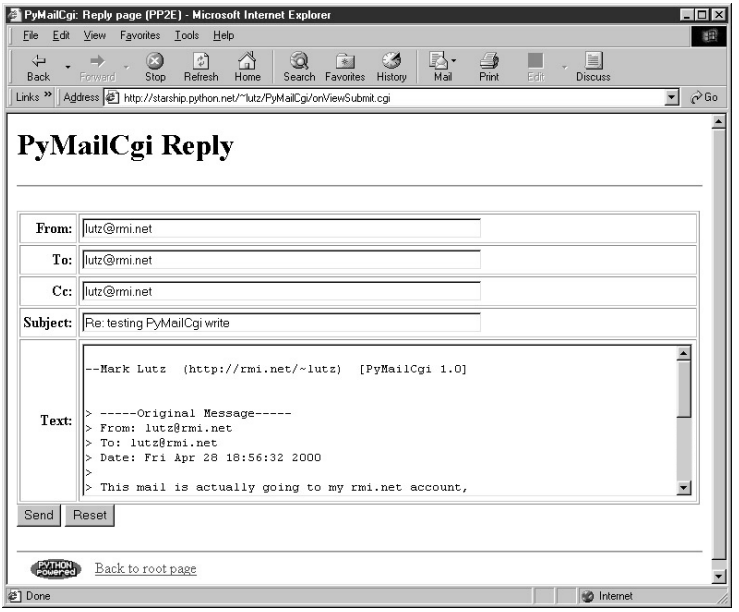


Рис. 13.16. Страница ответа PyMailCgi

Операции переадресации осуществляются практически так же, за исключением некоторых отличий в заголовках сообщений. Всю эту работу мы получаем «бесплатно», так как страницы Reply и Forward генерируются при вызове `commonhtml.editpage` – той же утилиты, с помощью которой создается страница для составления нового сообщения. Мы просто передаем здесь этой утилите готовые строки заголовков (например, при ответе к тексту темы добавляется «Re:»). Такого же рода прием с повторным ис-

пользованием применялся в PyMailGui, но в ином контексте. В PyMailCgi один сценарий обрабатывает три страницы; в PyMailGui одна функция обратного вызова обрабатывает три кнопки, но архитектура аналогична.

Удаление

В результате выбора действия Delete в странице просмотра сообщения и нажатия кнопки Next сценарий `onViewSubmit` немедленно удаляет просматриваемое сообщение. Удаления осуществляются путем вызова повторно используемой вспомогательной функции удаления, код которой представлен в примере 11.18; вызов этой утилиты заключен в вызов `commonhtml.runsilent`, который предотвращает вывод данных операторами `print` в ответный поток HTML (они являются лишь сообщениями о статусе, а не кодом HTML). На рис. 13.17 показана операция удаления в действии.

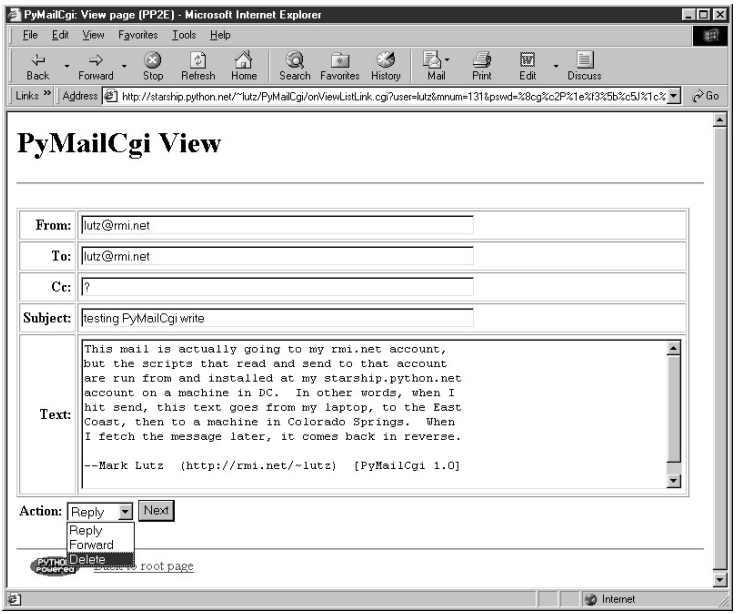


Рис. 13.17. Страница просмотра сообщений PyMailCgi, выбрано удаление

Как отмечалось, Delete является единственной операцией, использующей данные учетной записи POP (пользователь, пароль и сайт), переданные из скрытых полей предыдущей страницы (просмотра сообщения). Напротив, операции Reply и Forward форматируют страницу редактирования, которая в конечном счете отошлет сообщение серверу SMTP; никакие данные POP не требуются и не передаются. Но к этому моменту пароль POP накрутил в своем перемещении не одну милю. В действительности он мог пройти по телефонным линиям, спутниковым каналам связи и целым континентам, путешествуя с машины на машину. Этот процесс проиллюстрирован ниже:

1. Ввод (клиент): Пароль начинает свою жизнь с ввода на странице регистрации у клиента (или помещения в явный URL), в незашифрованном виде. При вводе в форму в веб-броузере каждый символ показывается в виде звездочки (*).

2. Загрузка списка сообщений (от клиента через CGI-сервер на POP-сервер): Затем он передается от клиента на CGI-сервер, который пересылает его вашему POP-серве-

ру для загрузки списка почтовых сообщений. Клиент посылает пароль в незашифрованном виде.

3. Строки URL на странице списка (CGI-сервер клиенту): Для управления поведением следующего сценария пароль встраивается в саму страницу со списком для выбора почтовых сообщений в виде параметров URL гиперссылок, зашифрованный и в кодировке URL.
4. Загрузка сообщения (от клиента через сервер CGI на сервер POP): Когда сообщение выбирается из списка, пароль посылается следующему сценарию, указанному в URL; сценарий CGI расшифровывает его и посылает серверу POP для загрузки выбранного сообщения.
5. Поля страницы просмотра (сервер CGI клиенту): Для управления поведением следующего сценария пароль встраивается в саму страницу просмотра в виде скрытых полей ввода HTML, в зашифрованном виде и кодировке HTML.
6. Удаление (от клиента через сервер CGI на сервер POP): Наконец, пароль снова передается от клиента на сервер CGI, на этот раз в виде значений скрытых полей формы; сценарий CGI расшифровывает его и посылает серверу POP для удаления сообщения.

Попутно сценарии передавали пароль между страницами как параметр URL или скрытое поле ввода HTML; в том и другом случае всегда передается зашифрованная строка, и никогда в одной операции не передаются одновременно незашифрованный пароль и имя пользователя. При запросе удаления перед передачей серверу POP пароль должен быть расшифрован с помощью модуля `secret`. Если сценарий смог снова обратиться к серверу POP и удалить выбранное сообщение, появляется еще одна страница подтверждения, которая показана на рис. 13.18.

Рис. 13.18. Подтверждение удаления в PyMailCgi

Обратите внимание, что после успешного удаления нужно действительно щелкнуть по ссылке «Back to root page» – не пользуйтесь в этот момент кнопкой броузера Back, чтобы вернуться в список выбора сообщений, потому что в результате удаления относительные номера некоторых сообщений в этом списке изменились. PyMailGui решал эту проблему, удаляя сообщения только при выходе, но PyMailCgi удаляет сообщения немедленно, потому что понятия «при выходе» в нем не существует. При щелчке по ссылке для просмотра в старой странице со списком может быть вызвано не то сообщение, которое вы предполагаете, если оно следует за сообщением, которое удалено.

Это свойство POP в целом: поступающая почта добавляется в список с более высокими номерами сообщений, но при удалении почта изымается из произвольного места списка, и потому изменяются номера всех сообщений, следующих за удаленными. Даже в PyMailGui номера некоторых сообщений могут оказаться неверными, если удалить почту другой программой в то время, когда открыт GUI (например, во втором экземпляре PyMailGui). В другом варианте обе почтовые программы могут удалить почту с сервера сразу после ее загрузки, чтобы удаление не оказывало воздействия на идентификаторы POP (такую схему использует, например, Microsoft Outlook), но при этом необходимы дополнительные механизмы для сохранения удаленной почты, позволяющие обращаться к ней в дальнейшем.

Одна тонкость: для ответа и переадресации сценарий действий с почтой `onViewSubmit` цитирует исходное сообщение с добавлением `>` перед строками и вставкой исходных строк заголовков «From:», «To:» и «Date:» перед исходным текстом сообщения. Обратите, однако, внимание, что заголовки исходных сообщений берутся из формы ввода CGI, а не из синтаксического анализа исходного сообщения (в тот момент сообщение не доступно непосредственно). Иными словами, сценарий берет значения заголовков из полей ввода формы на странице просмотра. Так как поля «Date:» на странице просмотра нет, дата исходного сообщения также передается сценарию действий с сообщением в виде скрытого поля ввода, чтобы не загружать сообщение заново. Попробуйте проследить по коду в листингах этой главы, как даты переходят с одной страницы на другую.

Вспомогательные модули

В этом разделе представлен исходный код вспомогательных модулей, импортируемых и используемых показанными выше сценариями страниц. Здесь вы не увидите новых снимков экранов, потому что это утилиты, а не сценарии CGI (обратите внимание на их расширения *.py*). Кроме того, изучать эти модули в отдельности не слишком полезно, и они помещены здесь, в основном, для справки при разборе кода сценариев CGI. Дополнительные детали, которые здесь не повторены, можно найти в предыдущих материалах данной главы.

Внешние компоненты

При установке PyMailCgi и других выполняемых на сервере программ этой книги я просто загружаю содержимое каталога примеров *Cgi-Web* со своего лэптопа в веб-каталог верхнего уровня моей учетной записи на сервере (*public_html*). Каталог *Cgi-Web* есть также на CD, прилагаемом к этой книге, и является копией того, который находится у меня на PC. Я не копирую на свой веб-сервер весь дистрибутив примеров книги, потому что код вне каталога *Cgi-Web* не предназначен для выполнения на веб-сервере.

Однако, впервые установив PyMailCgi, я столкнулся с проблемой: он использует модули, код которых описан в других частях книги и потому находится в каталогах за пределами *Cgi-Web*. Например, он повторно использует модули `mailconfig` и `pymail`, написанные в главе 11, но оба они находятся вне каталога примеров CGI. В таких внешних зависимостях обычно нет ничего страшного, при условии импорта пакетов или надлежащим образом настроенной `sys.path` при запуске. Однако в контексте CGI-сценариев важно, что имеющееся у меня на машине разработчика может отсутствовать на машине веб-сервера, где установлены сценарии.

Чтобы справиться с этой проблемой (и не загружать при этом на мой веб-сервер полный дистрибутив примеров книги), я определил на верхнем уровне *Cgi-Web* каталог с именем *Extern*, в который по мере надобности копируются все необходимые внешние

модули. Для данной системы в *Extern* входит подкаталог с названием *Email*, куда скопированы модули `mailconfig` и `pymail` для загрузки на сервер.

Наличие лишних копий файлов не идеально, но все это можно автоматизировать с помощью сценариев установки, которые автоматически копируют в *Extern* и затем загружают на сервер содержимое *Cgi-Web* через FTP с помощью модуля Python `ftplib` (обсуждавшегося в главе 11). На случай, если я решу изменить эту структуру, я заключил все обращения к внешним именам во вспомогательный модуль примера 13.10.

Пример 13.10. PP2E\Internet\Cgi-Web\PyMailCgi\externs.py

```
##################################################################################################
# Обособляет весь импорт модулей, находящихся вне каталога PyMailCgi. Обычно они находятся
# в PP2E.Internet.Email, но при установке PyMailCgi я копирую только содержимое каталога
# Cgi-Web в public_html на сервере, но на сервере нет каталога PP2E.
# Вместо этого я либо копирую импортируемые в этом файле модули в родительский каталог
# PyMailCgi или изменяю каталог, добавляемый здесь в путь поиска модулей sys.path.
# Поскольку все остальные модули берут внешние файлы отсюда, при их перемещении нужно
# сделать изменения только в одном месте. Можно утверждать, что это грубо,
# но я помещаю на машину сервера только код Интернета.
##################################################################################################

import sys
sys.path.append('..')                    # см. каталог, где на сервере установлен Email
from  Extern import Email                # предполагается каталог ../Extern с Email
from  Extern.Email import pymail         # можно использовать имена Email.pymail или pymail
from  Extern.Email import mailconfig
```

Этот модуль добавляет родительский каталог PyMailCgi в `sys.path`, чтобы сделать видимым каталог *Extern* (вспомните, что в PYTHONPATH может быть что угодно, когда сценарии CGI выполняются под пользователем «nobody»), и заранее импортирует все внешние имена, требуемые PyMailCgi, в его собственное пространство имен. Он также поддерживает возможные изменения в будущем: поскольку все внешние ссылки в PyMailCgi осуществляются через этот модуль, я должен изменить только один этот файл, если внешние компоненты позже будут установлены другим образом.

Для справки в примере 13.11 снова показана часть внешнего модуля `mailconfig`. Для PyMailCgi он копируется в *Extern* и может быть скорректирован, как нужно, на сервере (например, строка подписи в этом контексте несколько отличается). Посмотрите файл *pymail.py* в главе 11 и попробуйте в качестве упражнения написать сценарий автоматического копирования и загрузки на сервер каталога *Cgi-Web\Extern*; это не настолько сложно, чтобы заставить меня написать его самому.

Пример 13.11. PP2E\Internet\Cgi-Web\Extern\Email\mailconfig.py

```
##################################################################################
# Отсюда сценарии e-mail получают имена серверов: сделайте изменения,
# отражающие ваши имена машины/пользователя; можно было бы передавать их в командной строке
##################################################################################

# машина сервера SMTP (отправка электронной почты)
smtpservername = 'smtp.rmi.net'          # или starship.python.net, 'localhost'

# машина сервера POP3, пользователь (загрузка)
popservername  = 'pop.rmi.net'           # или starship.python.net, 'localhost'
popusername    = 'lutz'                  # при запуске запрашивается пароль

...остальное опущено

# личные данные, используемые PyMailGui при заполнении форм;
```

```
# sig – может быть блоком в тройных кавычках, игнорируется для пустой строки;
# addr – используется в качестве начального значения поля "From", если не пуст

myaddress   = 'lutz@rmi.net'
mysignature = '--Mark Lutz  (http://rmi.net/~lutz)  [PyMailCgi 1.0]'
```

Интерфейс почты POP

Вспомогательный модуль `loadmail` примера 13.12 зависит от внешних файлов и заключает в себе доступ к почте на удаленном POP-сервере. В настоящее время он экспортирует одну функцию, `loadnewmail`, которая возвращает список всех сообщений на заданной учетной записи POP; вызвавшему ее неизвестно, загружается ли эта почта из Сети, находится ли в памяти или загружается из постоянного хранилища на машине сервера CGI. Так сделано намеренно – изменения в `loadmail` не оказывают влияния на его клиентов.

Пример 13.12. PP2E\Internet\Cgi-Web\PyMailCgi\loadmail.py

```
################################################################################
# Загрузчик списка почты; на будущее – изменить, чтобы сохранять список почты в промежутке
# между запусками сценария cgi и не перегружать каждый раз заново всю почту;
# если сделать правильно, клиенты, использующие эти интерфейсы, ничего не заметят;
# пока для простоты при каждой операции перегружается вся почта
################################################################################

from commonhtml import runsilent         # подавить команды print (без флага verbose)
from externs    import Email

# загрузить все письма, начиная с номера 1
# может возбудить исключительную ситуацию

def loadnewmail(mailserver, mailuser, mailpswd):
    return runsilent(Email.pymail.loadmessages, (mailserver, mailuser, mailpswd))
```

Особо интересного здесь ничего нет – просто интерфейс и обращения к другим модулям. Функция `Email.pymail.loadmessages` (из главы 11, повторно используется) с помощью модуля Python `poplib` загружает почту через сокеты. Все эти действия заключены в функцию `commonhtml.runsilent` с целью не выводить сообщения `pymail` в ответный поток HTML (хотя все исключительные ситуации `pymail` распространяются дальше).

Однако в существующем виде `loadmail` загружает всю входящую почту, чтобы создать страницу со списком выбора, и заново перезагружает всю почту после получения сообщения из списка. Такая схема крайне неэффективна, если на сервере есть много почты. Я отмечал задержки порядка десятков секунд, когда почтовый ящик полон. С другой стороны, серверы могут быть медленными в целом, поэтому дополнительное время, занимаемое перезагрузкой почты, не всегда имеет значение – я был свидетелем аналогичных задержек на сервере и для пустых почтовых ящиков и для простых страниц HTML.

Более важно, что `loadmail` предназначен служить только в качестве почтового интерфейса в первом приближении – чего-то вроде рабочего прототипа. Если я буду развивать эту систему, она сможет легко буферизовать загруженную почту, скажем, в файле, контейнере shelve или базе данных на сервере. Поскольку в интерфейсе, экспортируемом `loadmail`, не понадобится делать изменения при введении механизма буферизации, клиенты этого модуля смогут работать по-прежнему. Варианты создания хранилищ на сервере будут исследованы в следующей главе.

Шифрование паролей POP

Пора вызывать полицию. Выше мы обсуждали подход к защите паролей, принятый в PyMailCgi. Вкратце, он стремится не допустить передачи имени пользователя и пароля учетной записи POP через Сеть одновременно в одной операции, если только пароль не зашифрован с помощью модуля *secret.py* на сервере. Этот модуль может быть своим в каждом месте, где установлен PyMailCgi, и в любой момент может быть заменен – зашифрованные пароли не хранятся постоянно и существуют только на протяжении одного интерактивного сеанса обработки почты.[1] В примере 13.13 представлен модуль шифрования, который я установил на своем сервере при работе над этой книгой.

Пример 13.13. PP2E\Internet\Cgi-Web\PyMailCgi\secret.py

```
##################################################################################################
# PyMailCgi шифрует пароль pop, когда он пересылается клиенту или от него через сеть вместе
# с именем пользователя как скрытые текстовые поля или параметры явного url; использует функции
# encode/decode этого модуля для зашифрования пароля - загрузите на сервер собственную версию
# этого модуля, чтобы использовать другой механизм шифрования; pymail также не сохраняет пароль
# на сервере и не повторяет пароль при вводе, но это не дает 100% защиты – файл этого модуля сам
# может оказаться уязвим со стороны злоумышленников; примечание: в Python 1.6 модуль сокетов
# будет содержать стандартную (но необязательную) поддержку сервером сокетов openSSL
# для программирования на Python защищенных транзакций в Интернете;
# см. документацию по модулю socket 1.6;
##################################################################################################

forceReadablePassword = 0
forceRotorEncryption  = 1

import time, string
dayofweek = time.localtime(time.time())[6]

#############################################################################
# схемы шифрования строк
#############################################################################

if not forceReadablePassword:
    # по умолчанию не делать ничего: вызовы urllib.quote или cgi.escape в commonhtml.py
    # выполнят необходимую кодировку пароля для встраивания его в URL или HTML;
    # модуль cgi автоматически снимает кодировку;

    def stringify(old):   return old
    def unstringify(old): return old

else:
  # преобразование кодированной строки в/из строки числовых символов, чтобы избежать проблем
  # с некоторыми специальными/непечатаемыми символами, но сохранить возможность чтения
  # результата (хотя и зашифрованного); в некоторых броузерах есть проблемы
  # с преобразованными амперсандами и т. д.;

    separator = '-'

    def stringify(old):
        new = ''
        for char in old:
```

[1] Учтите, что есть другие способы обеспечения защиты пароля помимо индивидуальных схем шифрования, описываемых в этом разделе. Например, модуль Python socket поддерживает теперь серверную часть протокола защищенных сокетов OpenSSL. С его помощью сценарии могут возлагать задачу обеспечения защиты на веб-броузеры и серверы. С другой стороны, такие схемы не дали бы повода представить в этой книге стандартные средства Python для шифрования.

```
            ascii = str(ord(char))
            new   = new + separator + ascii       # '-ascii-ascii-ascii'
        return new

    def unstringify(old):
        new = ''
        for ascii in string.split(old, separator)[1:]:
            new = new + chr(int(ascii))
        return new

##############################################################################
# схемы шифрования
##############################################################################

if (not forceRotorEncryption) and (dayofweek % 2 == 0):
    # использовать свою схему в дни с четными номерами (0=понедельник)
    # предостережение: возможен сбой при переходе суток

    def do_encode(pswd):
        res = ''
        for char in pswd:
            res = res + chr(ord(char) + 1)        # добавить 1 к каждому коду ascii
        return str(res)

    def do_decode(pswd):
        res = ''
        for char in pswd:
            res = res + chr(ord(char) - 1)
        return res

else:
    # использовать для шифрования пароля модуль rotor стандартной библиотеки
    # это более надежное шифрование, чем в приведенном выше коде

    import rotor
    mykey = 'pymailcgi'

    def do_encode(pswd):
        robj = rotor.newrotor(mykey)              # использовать шифрование enigma
        return robj.encrypt(pswd)

    def do_decode(pswd):
        robj = rotor.newrotor(mykey)
        return robj.decrypt(pswd)

##############################################################################
# точки входа верхнего уровня
##############################################################################

def encode(pswd):
    return stringify(do_encode(pswd))             # шифрование плюс кодировка строки

def decode(pswd):
    return do_decode(unstringify(pswd))
```

В этом модуле шифрования реализованы две альтернативные схемы: простое отображение кодов символов ASCII и шифрование в стиле Enigma с помощью стандартного модуля `rotor`. Модуль `rotor` реализует сложный алгоритм шифрования, основанный на шифровальной машине «Enigma», использовавшейся нацистами для шифрования сообщений во время Второй мировой войны. Однако не пугайтесь: модуль Python `rotor` значительно труднее расколоть, чем нацистскую машинку!

Кроме шифрования этот модуль реализует метод *кодировки* для уже зашифрованных строк. По умолчанию функции кодировки ничего не делают, и система использует

обычную кодировку URL. Может быть выбрана схема кодировки, переводящая зашифрованную строку в строку цифр кода ASCII, разделенных черточками. Оба метода кодировки делают непечатные символы зашифрованной строки печатными.

Схема шифрования, используемая по умолчанию: rotor

Для иллюстрации проверим средства этого модуля в интерактивном режиме. Сначала поэкспериментируем со стандартным модулем Python `rotor`, лежащим в основе используемой по умолчанию схемы шифрования. Импортируем модуль, создаем новый объект `rotor` с ключом (дополнительно можно задать количество роторов) и вызываем методы шифрования и расшифрования:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python
>>> import rotor
>>> r = rotor.newrotor('pymailcgi')        # (ключ, [,число_роторов])
>>> r.encrypt('abc123')                    # может возвратить непечатаемые символы
' \323an\021\224'

>>> x = r.encrypt('spam123')               # результат той же длины, что и вход
>>> x
'* _\344\011pY'
>>> len(x)
7
>>> r.decrypt(x)
'spam123'
```

Обратите внимание, что один и тот же объект ротора может зашифровывать несколько строк, что результат может содержать непечатаемые символы (выводимые как escape-коды `\ascii`, возможно, в восьмеричном виде) и что результат всегда совпадает по длине с исходной строкой. Самое главное, что строка, зашифрованная с помощью `rotor`, может быть расшифрована в другом процессе (например, позднее другим сценарием CGI), если объект ротора создать заново:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python
>>> import rotor
>>> r = rotor.newrotor('pymailcgi')        # может быть расшифрована в новом процессе
>>> r.decrypt('* _\344\011pY')             # два символа в виде кодов ¨\ascii¨
'spam123'
```

Наш модуль `secret` по умолчанию использует для шифрования только `rotor` и никакой собственной кодировки. Он полагается на кодировку URL при встраивании пароля в параметр URL и кодировку HTML при встраивании пароля в скрытые поля форм. Для URL осуществляются следующие типы вызовов:

```
>>> from secret import encode, decode
>>> x = encode('abc$#<>&+')                 # это делают сценарии CGI (rotor)
>>> x
' \323a\016\317\326\023\0163'

>>> import urllib                           # это делает urllib.urlencode
>>> y = urllib.quote_plus(x)
>>> y
'+%d3a%0e%cf%d6%13%0e3'

>>> a = urllib.unquote_plus(y)              # это делает cgi.FieldStorage
>>> a
' \323a\016\317\326\023\0163'

>>> decode(a)                               # это делают сценарии CGI (rotor)
'abc$#<>&+'
```

Альтернативные схемы шифрования

Чтобы показать, как нужно писать альтернативные шифраторы и кодировщики, в secret включены также функция кодировки, возвращающая строку цифр, и шифратор перестановки кодов символов; они могут быть включены с помощью глобальных флагов переменных в начале модуля:

forceReadablePassword

При установке в *true* зашифрованный пароль преобразуется в строку цифр ASCII-кодов, разделяемых дефисами. По умолчанию имеет значение *false*, что возвращает к кодировкам URL и HTML.

forceRotorEncryption

При значении *false* в четные дни недели вместо rotor используется простое шифрование кодов символов. По умолчанию *true*, что вынуждает роторное шифрование.

Чтобы продемонстрировать работу этих альтернативных схем, установим forceReadablePassword в 1 и forceRotorEncryption в 0 и заново выполним импорт. Обратите внимание на то, что это глобальные переменные, которые должны устанавливаться *перед* импортом (или перезагрузкой) модуля, потому что они управляют выбором альтернативных операторов def. Только одна версия каждого вида функций создается в этом модуле:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python
>>> from secret import *
>>> x = encode('abc$#<>&+')
>>> x
'-98-99-100-37-36-61-63-39-44'

>>> y = decode(x)
>>> y
'abc$#<>&+'
```

На самом деле это происходит в два этапа – сначала шифрование, затем кодировка (функции верхнего уровня encode и decode сочетают в себе эти два этапа). Вот как это выглядит при поэтапном исполнении:

```
>>> t = do_encode('abc$#<>&+')          # только наше шифрование
>>> t
"bcd%$=?',"
>>> stringify(t)                        # добавить собственную кодировку
'-98-99-100-37-36-61-63-39-44'

>>> unstringify(x)                      # снять кодировку
"bcd%$=?',"
>>> do_decode(unstringify(x))           # обратить оба этапа
'abc$#<>&+'
```

В этой альтернативной схеме шифрования просто прибавляется 1 к значению ASCII-кода каждого символа, а кодировщик помещает в результат целые числа ASCII-кодов. Можно также объединить шифрование rotor с нашей собственной кодировкой (установить обе переменные forceReadablePassword и forceRotorEncryption в 1), но кодировка URL в urllib работает не хуже. Вот пример действия различных схем; перед каждой перезагрузкой *secret.py* редактируется и сохраняется:

```
>>> import secret
>>> secret.encode('spam123')          # по умолчанию: rotor, без дополнительной кодировки
'* _\344\011pY'

>>> reload(secret)                    # forcereadable=1, forcerotor=0
<module 'secret' from 'secret.py'>
```

```
>>> secret.encode('spam123')
'-116-113-98-110-50-51-52'

>>> reload(secret)                    # forcereadable=1, forcerotor=1
<module 'secret' from 'secret.py'>
>>> secret.encode('spam123')
'-42-32-95-228-9-112-89'
>>> ord('Y')                          # последний символ действительно 'Y'
89

>>> reload(secret)                    # вернуться к rotor по умолчанию, без stringify
<module 'secret' from 'secret.pyc'>
>>> import urllib
>>> urllib.quote_plus(secret.encode('spam123'))
'%2a+_%e4%09pY'
>>> 0x2a                              # первый символ действительно 42, '*'
42
>>> chr(42)
'*'
```

Можно задать в своем *secret.py* любой тип шифрования и кодировки, если он придерживается ожидаемого протокола – функции шифрования и расшифрования должны получать и возвращать строки. Можно чередовать схемы по дням недели, как это делается здесь (однако учтите возможность сбоя, если система используется во время перехода стрелки часов через полночь!), и т. д. Несколько заключительных замечаний:

Другие средства шифрования в Python

Есть другие средства шифрования для Python, поставляемые вместе с ним или имеющиеся в Сети; смотрите подробности на *http://www.python.org* и в руководстве по библиотеке. К некоторым схемам шифрования относятся серьезно, и они могут быть запрещены законом для экспорта, но эти правила со временем меняются.

Поддержка защищенных сокетов

Как уже говорилось, в Python 1.6 (еще не выпущенном, когда писалась книга) в модуле `socket` стандартная поддержка защищенных сокетов OpenSSL. OpenSSL является «open source»-реализацией протокола защищенных сокетов (загружать и устанавливать его нужно отдельно от Python – смотрите *http://www.openssl.org*). Там, где можно использовать этот протокол, он обеспечит лучшее и менее ограничивающее решение для защиты такой информации, как пароли, нежели ручная схема, принятая нами здесь.

Например, защищенные сокеты позволяют вводить и передавать имя пользователя и пароль с одной веб-страницы, что означает поддержку чтения почты произвольными пользователями. Лучшее, что мы можем предложить в отсутствие защищенных сокетов, это избегать смешивания незашифрованных значений имени пользователя и пароля и допустить, чтобы некоторые данные учетной записи и шифраторы располагались на сервере (как сделано здесь), или создавать две разные страницы ввода или два URL (для каждого из двух значений). Ни та, ни другая схемы не отличаются таким дружественным к пользователю отношением, как в подходе с защищенными сокетами. Большинство броузеров уже поддерживает SSL; за указаниями по добавлению его к Python на вашем сервере следует обратиться к руководству по библиотеке Python 1.6 (или выше).

Интернет-безопасность представляет собой слишком обширную тему, чтобы обращаться здесь к ней в полном объеме, и мы лишь поверхностно коснулись ее. За дополнительными сведениями по вопросам безопасности обращайтесь к книгам, посвященным исключительно технологиям веб-программирования.

На моем сервере файл *secret.py* будет со временем изменен, на случай, если злоумышленники следят за сайтом книги. Кроме того, его исходный код нельзя посмотреть с помощью сценария CGI `getfile`, написанного в главе 12 «Сценарии, выполняемые на сервере». Это означает, что если вы реально воспользуетесь этой системой, пароли в URL и скрытых полях форм могут выглядеть совершенно иначе, чем в этой книге. Мой пароль тоже будет изменен к тому времени, когда вы читаете эти слова, иначе можно было бы узнать мой пароль просто из этой книги!

Общий вспомогательный модуль

Файл *commonhtml.py*, показанный в примере 13.14, служит «центральным вокзалом» этого приложения – его код используется почти всеми остальными файлами системы. По большей части он самодокументирован, и все, что я хотел сказать о нем, я уже произнес в связи со сценариями CGI, использующими его.

Однако я ничего не сказал о поддержке в нем *отладки*. Обратите внимание, что этот модуль назначает `sys.stderr` в `sys.stdout`, чтобы заставить текст сообщений Python об ошибках выводиться в броузере клиента (помните, что неперехваченные исключительные ситуации выводят свои подробности в `sys.stderr`). Иногда это действует в PyMailCgi, но не всегда – текст ошибки появляется на веб-странице, только если вызов `page_header` уже вывел преамбулу ответа. Если нужно увидеть все сообщения об ошибках, вызовите `page_header` (или выведите строки `Content-type:` вручную) перед тем, как выполнять какую-либо обработку. В этом модуле также определены функции, сбрасывающие в броузер массу информации окружения CGI (`dumpstatepage`) и служащие оболочками для вызовов функций, выводящих сообщения о статусе, чтобы их вывод не попадал в поток HTML (`runsilent`).

Я предоставляю читателю самому раскрыть оставшуюся в этом коде магию. А мы пойдем дальше, нам предстоит читать, обращаться за справками и повторно использовать.

Пример 13.14. PP2E\Internet\Cgi-Web\PyMailCgi\commonhtml.py

```
#!/usr/bin/python
################################################################################
# Генерация HTML стандартного заголовка страницы, списка и нижнего колонтитула; в этом файле
# изолированы детали, относящиеся к генерации html; выводимый здесь текст поступает клиенту
# через сокеты, создавая в веб-броузере части новой веб-страницы; использует по одному print
# на строку, а не блоки строк, использует urllib для кодировки параметров в ссылках url,
# создаваемых из словаря, но cgi.escape для вставки их в скрытые поля html; некоторые инструменты
# отсюда могут использоваться вне pymailcgi; можно было бы возвращать генерируемый html вместо
# вывода в поток, чтобы включать в другие страницы; можно было бы облечь в структуру единого
# сценария cgi, который получает и проверяет имя следующей операции как скрытое поле формы;
# предупреждение: система действует, но была написана в основном во время 2-часовой задержки
# в чикагском аэропорту O'Hare: некоторые компоненты стоило бы отшлифовать; для автономного
# выполнения на starship через командную строку введите "python commonhtml.py"; для автономного
# выполнения через удаленный веб-броузер переименуйте файл как .cgi и запустите fixcgi.py.
################################################################################

import cgi, urllib, string, sys
sys.stderr = sys.stdout                 # показывать сообщения об ошибках в броузере
from externs import mailconfig          # из пакета, находящегося на сервере

# корень моего адреса
```

```
urlroot = 'http://starship.python.net/~lutz/PyMailCgi'

def pageheader(app='PyMailCgi', color='#FFFFFF', kind='main', info=''):
    print 'Content-type: text/html\n'
    print '<html><head><title>%s: %s page (PP2E)</title></head>' % (app, kind)
    print '<body bgcolor="%s"><h1>%s %s</h1><hr>' % (color, app, (info or kind))

def pagefooter(root='pymailcgi.html'):
    print '</p><hr><a href="http://www.python.org">'
    print '<img src="../PyErrata/PythonPoweredSmall.gif" '
    print 'align=left alt="[Python Logo]" border=0 hspace=15></a>'
    print '<a href="%s">Back to root page</a>' % root
    print '</body></html>'

def formatlink(cgiurl, parmdict):
    """
    создать ссылку запроса "%url?key=val&key=val" из словаря;
    преобразует str() всех ключей и значений как %xx, заменяет ' ' на +
    обратите внимание на разницу кодировок url и html (cgi.escape)
    """
    parmtext = urllib.urlencode(parmdict)           # вызывает urllib.quote_plus
    return '%s?%s' % (cgiurl, parmtext)             # всю работу делает urllib

def pagelistsimple(linklist):                       # показать простой упорядоченный список
    print '<ol>'
    for (text, cgiurl, parmdict) in linklist:
        link = formatlink(cgiurl, parmdict)
        text = cgi.escape(text)
        print '<li><a href="%s">\n    %s</a>' % (link, text)
    print '</ol>'

def pagelisttable(linklist):                        # показать список в таблице
    print '<p><table border>'                       # для верности выполнить перекодировку
    count = 1
    for (text, cgiurl, parmdict) in linklist:
        link = formatlink(cgiurl, parmdict)
        text = cgi.escape(text)
        print '<tr><th><a href="%s">View</a> %d<td>\n %s' % (link, count, text)
        count = count+1
    print '</table>'

def listpage(linkslist, kind='selection list'):
    pageheader(kind=kind)
    pagelisttable(linkslist)         # [('text', 'cgiurl', {'parm':'value'})]
    pagefooter()

def messagearea(headers, text, extra=''):
    print '<table border cellpadding=3>'
    for hdr in ('From', 'To', 'Cc', 'Subject'):
        val = headers.get(hdr, '?')
        val = cgi.escape(val, quote=1)
        print '<tr><th align=right>%s:' % hdr
        print '    <td><input type=text '
        print '    name=%s value="%s" %s size=60>' % (hdr, val, extra)
    print '<tr><th align=right>Text:'
    print '<td><textarea name=text cols=80 rows=10 %s>' % extra
    print '%s\n</textarea></table>' % (cgi.escape(text) or '?')   # if has </>s

def viewpage(msgnum, headers, text, form):
    """
    по View + выбор (щелчок по созданной ссылке)
    очень тонкое место: в это время pswd был в ссылке в url-кодировке и затем раскодировался
```

```
    парсером входа cgi; здесь он встроен в html, поэтому применяем cgi.escape; обычно
    появляются непечатные символы в html скрытых полей, но в ie и ns как-то работает:
    в url:  ?user=lutz&mnum=3&pswd=%8cg%c2P%1e%f0%5b%c5J%1c%f3&...
    в html: <input type=hidden name=pswd value="...непечатные..">
    можно бы urllib.quote поле html, но необходимо urllib.unquote
    в следующем сценарии (что не дает передать данные в URL, а не форме);
    можно вернуться к цифровому формату строк из secret.py
    """
    pageheader(kind='View')
    user, pswd, site = map(cgi.escape, getstandardpopfields(form))
    print '<form method=post action="%s/onViewSubmit.cgi">' % urlroot
    print '<input type=hidden name=mnum value="%s">' % msgnum
    print '<input type=hidden name=user value="%s">' % user     # из страницы|url
    print '<input type=hidden name=site value="%s">' % site     # для удалений
    print '<input type=hidden name=pswd value="%s">' % pswd     # пароль зашифрован
    messagearea(headers, text, 'readonly')

    # onViewSubmit.quotetext требует передачи даты страницей
    print '<input type=hidden name=Date value="%s">' % headers.get('Date','?')
    print '<table><tr><th align=right>Action:'
    print '<td><select name=action>'
    print '    <option>Reply<option>Forward<option>Delete</select>'
    print '<input type=submit value="Next">'
    print '</table></form>'                       # 'reset' здесь не требуется
    pagefooter()

def editpage(kind, headers={}, text=''):
    # no Send, View+select+Reply, View+select+Fwd
    pageheader(kind=kind)
    print '<form method=post action="%s/onSendSubmit.cgi">' % urlroot
    if mailconfig.mysignature:
        text = '\n%s\n%s' % (mailconfig.mysignature, text)
    messagearea(headers, text)
    print '<input type=submit value="Send">'
    print '<input type=reset  value="Reset">'
    print '</form>'
    pagefooter()

def errorpage(message):
    pageheader(kind='Error')                          # или sys.exc_type/exc_value
    exc_type, exc_value = sys.exc_info()[:2]          # но безопаснее, с учетом потоков
    print '<h2>Error Description</h2><p>', message
    print '<h2>Python Exception</h2><p>',  cgi.escape(str(exc_type))
    print '<h2>Exception details</h2><p>', cgi.escape(str(exc_value))
    pagefooter()

def confirmationpage(kind):
    pageheader(kind='Confirmation')
    print '<h2>%s operation was successful</h2>' % kind
    print '<p>Press the link below to return to the main page.</p>'
    pagefooter()

def getfield(form, field, default=''):
    # emulate dictionary get method
    return (form.has_key(field) and form[field].value) or default

def getstandardpopfields(form):
    """
    поля могут отсутствовать, быть '' или иметь реальные значения,
    прошитые в url; по умолчанию настройки mailconfig
```

```
    """
    return (getfield(form, 'user', mailconfig.popusername),
            getfield(form, 'pswd', '?'),
            getfield(form, 'site', mailconfig.popservername))

def getstandardsmtpfields(form):
    return  getfield(form, 'site', mailconfig.smtpservername)

def runsilent(func, args):
    """
    выполнить функцию без записи в stdout
    пример: подавление print в импортированных утилитах, чтобы вывод не попал клиенту/броузеру
    """
    class Silent:
        def write(self, line): pass
    save_stdout = sys.stdout
    sys.stdout  = Silent()                  # отправить вывод в фиктивный объект
    try:                                    # с методом write
        result = apply(func, args)          # попытка вернуть результат func
    finally:                                # всегда восстановить stdout
        sys.stdout = save_stdout
    return result

def dumpstatepage(exhaustive=0):
    """
    для отладки: вызвать в начале cgi для создания
    новой страницы с деталями состояния cgi
    """
    if exhaustive:
        cgi.test()                          # показать страницу с формой, окружением и т. д.
    else:
        pageheader(kind='state dump')
        form = cgi.FieldStorage()           # показать только имена/значения полей формы
        cgi.print_form(form)
        pagefooter()
    sys.exit()

def selftest(showastable=0):                # сделать фальшивую веб-страницу
    links = [                               # [(text, url, {parms})]
        ('text1', urlroot + '/page1.cgi', {'a':1}),
        ('text2', urlroot + '/page1.cgi', {'a':2, 'b':'3'}),
        ('text3', urlroot + '/page2.cgi', {'x':'a b', 'y':'a<b&c', 'z':'?'}),
        ('te<>4', urlroot + '/page2.cgi', {'<x>':'', 'y':'<a>', 'z':None})]
    pageheader(kind='View')
    if showastable:
        pagelisttable(links)
    else:
        pagelistsimple(links)
    pagefooter()

if __name__ == '__main__':                  # при запуске, а не импорте
    selftest(len(sys.argv) > 1)             # html выводится в stdout
```

Недостатки и преимущества сценариев CGI

Как было показано в этой главе, PyMailCgi все еще находится в стадии развития, но действует, как обещано: направив броузер на URL главной страницы, я могу проверить почту и отослать ее из любого места, где окажусь, если только смогу найти машину с веб-броузером. Подходят практически любая машина и броузер: даже не обя-

зательно должен быть установлен Python.[1] Для программы-клиента PyMailGui, которую мы написали в главе 11, это не так.

Но прежде чем бросаться всем в коллективное использование Интернета и совершенно позабыть традиционные API типа Tkinter, надлежит сказать несколько слов о ситуации в целом. Помимо общей иллюстрации более крупных приложений CGI этот пример был выбран, чтобы выделить некоторые компромиссы, встречающиеся при разработке приложений для веб. PyMailGui и PyMailCgi выполняют примерно одинаковые функции, но в корне отличны по реализации:

- PyMailGui является традиционной программой с интерфейсом пользователя: она целиком выполняется на локальной машине, для реализации интерфейсов вызывает функции библиотеки GUI API, выполняемой в том же процессе, и связывается с Интернетом через сокеты только при необходимости (например, для загрузки или отправки почты по требованию). Запросы пользователя немедленно направляются функциям обработчиков обратного вызова или методам, выполняемым локально, а между запросами состояние автоматически сохраняется в совместно используемых переменных. Например, PyMailGui загружает почту только один раз, хранит ее в памяти и получает вновь прибывшие сообщения только при следующей загрузке, потому что его память сохраняется в промежутке между событиями.

- PyMailCgi, как все системы CGI, состоит из сценариев, располагающихся и выполняемых на сервере и генерирующих HTML, чтобы взаимодействовать с пользователем через веб-броузер на машине клиента. Выполнение происходит только в контексте броузера, а запросы пользователя обрабатываются путем удаленного выполнения сценариев CGI на сервере. Если не ввести дополнительно действительную базу данных, каждый обработчик запроса будет выполняться автономно и только с той информацией о состоянии, которая явно передается от предыдущих состояний как скрытые поля форм или параметры URL. В настоящем виде PyMailCgi должен заново загружать всю почту, когда ему требуется каким-либо образом обработать входящую почту.

На базовом уровне в обеих системах для получения и отправки почты через сокеты используются модули Python POP и SMTP. Но в представляемых ими альтернативах реализации есть важные разветвления, о которых следует знать при создании систем для веб:

Издержки производительности

Сети тормозят сильнее, чем CPU. В настоящей реализации скорость или полнота PyMailCgi не идут ни в какое сравнение с PyMailGui. В PyMailCgi каждый раз, когда пользователь щелкает по кнопке передачи, запрос передается через сеть. Более точно, каждый запрос пользователя влечет расходы, связанные с передачей по сети, каждый обработчик обратного вызова (обычно) оборачивается порождением на сервере нового процесса, параметры поступают в виде текстовых строк, требующих синтаксического разбора, а отсутствие на сервере информации о состоянии при переходе к новой странице означает, что почту приходится часто перезагружать. Напротив, действия пользователя в PyMailGui запускают вызовы функций в том же процессе вместо передачи по сети и ветвления процессов, а состояние легко сохраняется в Python как переменные процесса (например, список загруженной

[1] Это свойство оказывается особенно полезным при посещении правительственных учреждений, в которых обычно можно получить доступ к веб-броузеру, но административные функции и более широкие возможности сетевых соединений предоставляются только системным администраторам с официальным допуском (и иностранным шпионам).

почты сохраняется между щелчками). Даже при сверхбыстром соединении с Интернетом CGI-система на сервере медленнее, чем программа на стороне клиента.[1]

Некоторые из этих узких мест могут быть устранены ценой увеличения сложности программы. Например, некоторыми веб-серверами используются потоки и пулы процессов, чтобы свести к минимуму создание процессов для сценариев CGI. Кроме того, часть информации о состоянии можно передавать вручную со страницы на страницу в скрытых полях формы и параметрах, генерируемых URL, а в промежутке между страницами состояние можно сохранять в базе данных, допускающей одновременный доступ, чтобы свести к минимуму перезагрузку почты (см. конкретный пример PyErrata в главе 14). Но нельзя пройти мимо того факта, что переправка событий сценариям через сеть происходит значительно медленнее, чем прямой вызов функций Python.

Издержки сложности

HTML неудобен. Поскольку PyMailCgi должен генерировать HTML, чтобы взаимодействовать с пользователем через веб-броузер, он более сложен (или, по крайней мере, менее читаем), чем PyMailGui. В некотором смысле сценарии CGI встраивают код HTML в Python. Так как в конечном результате получается смесь двух очень разных языков, создание интерфейса с помощью HTML в сценарии CGI может оказаться далеко не таким простым делом, как вызовы API GUI, например Tkinter.

Посмотрите, скажем, сколько труда мы приложили для кодировки HTML и URL в примерах этой главы; такие ограничения заложены в природе HTML. Кроме того, при изменении системы с целью сохранения состояния списка загруженной почты в базе данных в промежутке между страницами сложность системы, основанной на CGI, возрастет еще больше. Защищенные сокеты (например, OpenSSL в будущей версии Python 1.6) должны устранить затраты на шифрование вручную, но с ними появятся другие накладные расходы.

Издержки функциональности

На HTML можно выразить немного. HTML служит переносимым способом определения простых страниц и форм, но слаб или бесполезен для описания более сложных интерфейсов пользователя. Поскольку сценарии CGI создают интерфейсы пользователя путем вывода HTML в броузер, они весьма ограничены в отношении конструкций, используемых интерфейсом пользователя.

Попробуйте, например, реализовать программу обработки изображений и анимации в виде сценариев CGI: HTML не пригоден за пределами заполняемых форм и простого взаимодействия. Это как раз то ограничение, для преодоления которого были придуманы апплеты Java – программы, которые хранятся на сервере, но загружаются по требованию для выполнения у клиента, а при наличии доступа к полноценному API GUI – для создания более богатых интерфейсов пользователя. Тем не менее, программам, строго ограниченным стороной сервера, внутренне присущи ограничения HTML. Например, сценарии анимации, которые мы написали в конце главы 8 «Обзор Tkinter, часть 2», находятся далеко за рамками возможностей сценариев на сервере.

Преимущества переносимости

Все что вам нужно – это броузер. Во всяком случае на стороне клиента. Так как PyMailCgi выполняется через Сеть, с ним можно работать на любой машине, где

[1] По правде, некоторые операции Tkinter тоже отправляются обрабатывающей их библиотеке Tcl в виде строк, которые нужно анализировать. Со временем это может измениться; но здесь производится противопоставление CGI-сценариев GUI-библиотекам в целом, а не реализации конкретной библиотеки.

есть веб-броузер, независимо от наличия на ней Python и Tkinter. Это значит, что Python должен быть установлен только на одном компьютере: машине веб-сервера, где фактически располагаются и выполняются сценарии. Если вы знаете, что у пользователей вашей системы есть броузер Интернета, установка проста.

Если вы вспомните, Python и Tkinter тоже весьма переносимы – они выполняются на всех главных оконных системах (X, Windows, Mac), но для выполнения клиентской программы Python/Tk, например PyMailGui, вам потребуются на машине клиента собственно Python и Tkinter. Иное дело с приложениями, построенными как сценарии CGI: они будут работать на Macintosh, Linux, Windows и любой другой машине, которая может каким-то образом выводить веб-страницы HTML. В этом смысле HTML становится своего рода переносимым языком API GUI в сценариях CGI, интерпретируемых веб-броузером. Вам даже не нужен исходный код или байт-коды самих сценариев CGI – они выполняются на удаленном сервере, существующем где-то в сети, а не на машине, где работает броузер.

Требования к выполнению

Но вам нужен броузер. То есть сама природа основанных на веб систем может сделать их бесполезными в некоторых средах. Несмотря на вездесущность Интернета, все еще есть масса приложений, выполняемых в отсутствие броузеров или доступа к Интернету. Возьмите, например, встроенные системы, системы реального времени или защищенные правительственные приложения. Хотя в интрасетях (локальных сетях без внешних соединений) веб-приложения могут иногда выполняться, но в недалекие времена мне приходилось работать не в одной компании, где у клиентов вообще отсутствовали веб-броузеры. С другой стороны, у таких клиентов бывает проще установить на локальных машинах системы типа Python, чем организовать поддержку внутренней или внешней сети.

Требования к администрированию

В действительности необходим также сервер. Основанные на CGI системы вообще нельзя писать без доступа к веб-серверу. Кроме того, хранение программ на централизованном сервере создает довольно существенные административные издержки. Попросту говоря, в чистой архитектуре клиент/сервер клиенты проще, но сервер становится ресурсом с критическим путем и потенциальным узким местом производительности. Если централизованный сервер выйдет из строя, то вы, ваши служащие и клиенты можете лишиться доверия. Кроме того, если достаточно много клиентов пользуется одновременно общим сервером, издержки в скорости основанных на веб систем становятся еще более явными. На самом деле можно утверждать, что смещение в сторону архитектуры веб-сервера является отчасти движением вспять – ко временам централизованных мэйнфрэймов и тупых терминалов. Какой бы путь мы ни выбрали, разгрузка сервера и распределение обработки между клиентскими машинами позволяет хотя бы частично преодолеть это узкое место обработки.

Так как же лучше всего строить приложения для Интернета – как клиентские программы, общающиеся с Сетью, или как выполняемые на сервере программы, жизнь которых проходит в Интернете? Естественно, однозначного ответа на этот вопрос нет, так как все зависит от особых ограничений каждого приложения. Более того, ответов может быть больше, чем здесь предложено; для большинства стандартных проблем CGI уже предложены стандартные решения. Например:

Решения для стороны клиента

Программы, выполняемые у клиента и на сервере, могут перемешиваться различными способами. Например, программы *апплетов* располагаются на сервере, но загружаются и выполняются как программы клиента, имея доступ к богатым биб-

лиотекам GUI (подробнее об апплетах при обсуждении JPython в главе 15). Другие технологии, например встраивание JavaScript или Python непосредственно в код HTML, тоже поддерживают выполнение на стороне клиента и более богатые возможности GUI; такие сценарии располагаются в HTML на сервере, но после загрузки выполняются у клиента и имеют доступ к компонентам броузера через открытую объектную модель (см. обсуждение «Расширения сценариев для веб в Windows» ближе к концу главы 15). Новые расширения динамического HTML (DHTML) предоставляют еще один вариант сценариев клиента для изменения веб-страниц после их создания. Все эти технологии клиентской стороны добавляют собственные сложности, но ослабляют некоторые ограничения, налагаемые обычным HTML.

Решения, сохраняющие состояние

Некоторые серверы веб-приложений (например, Zope, описанный в главе 15) обеспечивают естественную поддержку сохранения состояний в промежутке между страницами, предоставляя объектные базы данных с одновременным доступом. В некоторые из этих систем действительно входит составной частью база данных (например, Oracle и mySql); другие используют файлы или хранилища постоянных объектов Python с соответствующей блокировкой (как будет изучено в следующей главе). Сценарии могут также передавать информацию о состоянии через скрытые поля форм и параметры генерируемых URL, как делается в PyMailCgi, либо хранить ее на машине клиента с помощью стандартного протокола cookie.

Cookies представляют собой фрагменты информации, сохраняемые у клиента по запросу сервера. Cookie создается в результате отправки сервером клиенту специальных заголовков в ответном HTML (`Set-Cookie: name=value`). После этого доступ к нему в сценарии CGI производится как к значению особой переменной окружения (`HTTP_COOKIE`), содержащей данные cookie, загружаемые с клиента. Поищите на *http://www.python.org* дополнительные сведения об использовании cookies в сценариях Python, в том числе бесплатно распространяемый модуль *cookie.py*, автоматизирующий процесс трансляции cookie.[1] Cookies сложнее, чем переменные программы, и вызывают споры (некоторые считают их навязчивыми), но они могут снять некоторые простые задачи сохранения состояния.

Решения для генерации HTML

Дополнения (add-ons) тоже могут отчасти уменьшить сложность встраивания HTML в CGI-сценарии Python, хотя и ценой некоторого снижения скорости выполнения. Например, система HTMLgen, описываемая в главе 15, позволяет программам строить страницы как деревья объектов Python, которые «умеют» создавать HTML. При использовании системы такого типа сценарии Python имеют дело только с объектами, а не с синтаксисом самого HTML. Другие системы, например PHP и Active Server Pages (описываемые в той же главе), позволяют встраивать в HTML код на языке сценариев, который выполняется на сервере, чтобы динамически создавать или определять часть HTML, отправляемого клиенту в ответ на запрос.

Очевидно, технология Интернета предполагает некоторые компромиссы архитектуры и продолжает быстро развиваться. Тем не менее, это уместный контекст поставки многих (хотя и не всех) приложений. Как и во всяком проектном решении, вы сами должны быть судьей. Хотя при размещении систем в веб могут потеряться скорость, функциональность и увеличиться сложность, значение таких потерь со временем, вероятно, уменьшится.

---

[1] Смотрите также новый стандартный модуль `cookie` в выпуске Python 2.0.

14

Более крупные примеры сайтов, часть 2

«Случаются и опечатки»

В этой главе представлен второй из двух конкретных примеров веб-программирования на Python со стороны сервера. Здесь рассказывается о конструкции и реализации *PyErrata* – основанного на CGI сайта, целиком реализованного на Python и дающего пользователям возможность посылать комментарии и сообщения об ошибках, а также демонстрирующего идеи, лежащие в основе постоянного хранения в базах данных в мире CGI. Как мы увидим, этот конкретный пример учит созданию сценариев на стороне сервера и приемам разработки на Python.

Веб-сайт PyErrata

Предыдущая глава завершилась обсуждением недостатков размещения приложений в Сети. Но, сообщив о причинах, по которым нежелательна разработка таких систем, я хочу, совершенно противореча себе, представить систему, которая взывает об интернет-реализации. В этой главе представлен сайт PyErrata – программа Python, позволяющая любым людям на любой машине передавать через Сеть комментарии по поводу книги и сообщения об ошибках (обычно называемые *errata*) с помощью одного лишь броузера.

PyErrata в некоторых отношениях проще примера PyMailCgi, представленного в предыдущей главе. С точки зрения пользователя система PyErrata скорее иерархическая, чем линейная: взаимодействие с пользователем короче и порождает меньше страниц. Кроме того, в PyErrata мало данных состояния сохраняется в самих страницах; параметры URL передают состояние только в одном отдельном случае, и не генерируется скрытых полей форм.

С другой стороны, PyErrata вводит совершенно новое измерение: *постоянное хранилище данных* (*persistent data storage*). Состояние (сообщения об ошибках и комментарии) постоянно хранится этой системой на сервере, в плоских файлах или в базе данных, основанной на shelve. В обоих случаях вызывается призрак одновременного обновления, так как в одно и то же время обратиться к сайту может любое количество пользователей киберпространства.

Задачи системы

Чтобы вам не размышлять слишком долго над кажущимся парадоксом книги, вместе с которой поставляется собственная система сообщений об ошибках, я должен сообщить некоторые предпосылки этого. За последние пять лет мне посчастливилось написать четыре книги, большую главу справочника и ряд журнальных статей и учебных материалов. Изменения, происшедшие в мире Python, тоже дали возможность

переработать книги с самого основания. Это была чрезвычайно полезная и выгодная работа (ну, по крайней мере, полезная).

Но одним из первых настоящих уроков, которые человек получает, приобщаясь к издательской деятельности, является понимание неизбежности опечаток. В самом деле, как бы вы ни стремились к совершенству, в книгах будут случаться ошибки. Более того, в больших книгах оказывается больше ошибок, чем в маленьких, а в области технических публикаций читатели часто достаточно неплохо подготовлены и с удовольствием пошлют автору e-mail при обнаружении этих ошибок.

Это прекрасная штука, которая помогает авторам вычищать опечатки при повторных изданиях. Я всегда поощряю и ценю почту, получаемую от читателей. Но я получаю массу почты – иногда так много, что с учетом графика своей работы мне даже трудно ответить на каждое сообщение, не говоря уже о расследовании и действиях по каждому сообщению о типографской ошибке. Я получаю массу другой почты и могу пропустить по невнимательности сообщение читателя об опечатке.

Примерно год назад я понял, что не могу справиться со всем этим потоком сообщений, и задумался об альтернативах. Очевидным способом сокращения расходов на анализ отчетов является передача ответственности – переложение хотя бы части задач обработки отчетов на тех, кто генерирует эти отчеты. Это значит, что мне нужна общедоступная система, независимая от моей учетной записи почты, которая автоматизирует отправку отчетов и регистрирует отчеты, которые можно будет просмотреть, когда позволит время.

Конечно, как раз для такого рода потребностей подходит Интернет. При реализации системы сообщений об ошибках в виде сайта любой читатель может передать сообщение с любой машины, где есть броузер, независимо от того, установлен ли на ней Python. Кроме того, эти сообщения можно записывать в базу данных на сайте для последующего изучения автором или читателями, а не извлекать вручную из входящей почты.

Реализацией этих идей служит система PyErrata – сайт, реализованный с помощью программ Python, выполняемых на сервере. PyErrata дает пользователям возможность передавать отчеты об ошибках и комментарии о данном издании *Programming Python*, а также просматривать все переданные ранее сообщения, используя различные ключи сортировки. Задача в том, чтобы представить замену обычным страницам списка опечаток, которые мне приходилось в прошлом поддерживать вручную для других книг.

Более чем все другие, относящиеся к веб, примеры из этой книги, PyErrata показывает, сколько работы можно сэкономить с помощью небольших сценариев Интернета. Для поддержки первого издания этой книги я вручную отредактировал файл HTML с перечислением всех обнаруженных ошибок. С помощью PyErrata программы на стороне сервера генерируют такие страницы динамически из базы данных, заполненной пользователем. Поскольку страницы списков создаются по требованию, PyErrata не только публикует и автоматизирует создание списка, но также предоставляет несколько способов просмотра данных из сообщений. Список для первого издания в статическом файле HTML я даже не пытался переупорядочить.

PyErrata представляет собой определенный эксперимент в открытых системах и потому не защищена от злоупотреблений. Мне по-прежнему приходится вручную изучать отчеты, когда это позволяет время. Но у нее, по крайней мере, есть возможность облегчить одно из тех дел, о которых обычно не упоминается в типичных контрактах с издательствами.

Обзор реализации

Подобно другим базирующимся в веб системам из этой части книги, PyErrata состоит из совокупности файлов HTML, вспомогательных модулей Python и написанных на Python сценариев CGI, выполняемых на общедоступном сервере, а не у клиента. В отличие от тех других веб-систем, PyErrata также реализует постоянную базу данных и определяет дополнительные структуры каталогов для ее поддержки. На рис. 14.1 показано содержимое сайта верхнего уровня, как его видно в Windows из диалога PyEdit Open.

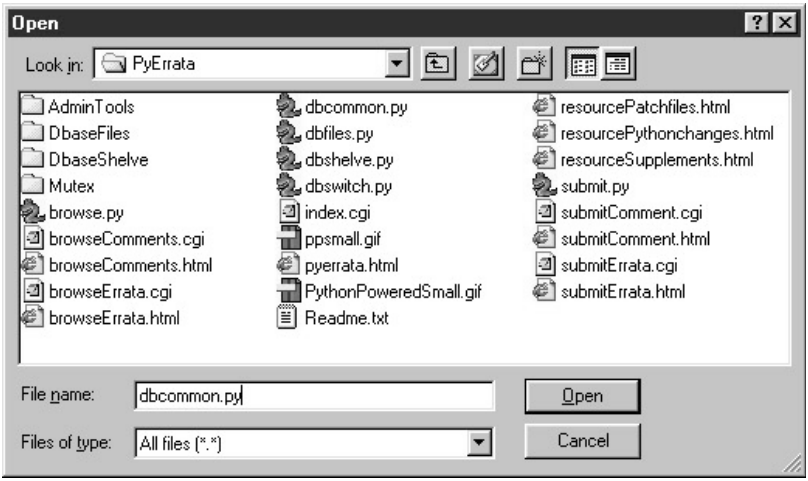


Рис. 14.1. Содержимое сайта PyErrata

Аналогичная структура есть на прилагаемом к книге CD-ROM. Для установки этого сайта в Сети все файлы и каталоги, которые вы здесь видите, загружены на машину сервера и записаны в подкаталог *PyErrata* корневого каталога, открытого для веб (моего каталога *public_html*). Файлы верхнего уровня на этом сайте реализуют операции просмотра и передачи, а также интерфейсы баз данных. В этом списке есть также несколько файлов страниц ресурсов и графики, но книга их обходит. Помимо файлов на этом сайте есть собственные подкаталоги:

- *Mutex* является пакетом Python, содержащим модуль утилит взаимного исключения для файлов shelve, а также тестовые сценарии для этой вспомогательной модели.
- *AdminTools* содержит сценарии системных утилит, выполняемых автономно из командной строки.
- *DbaseFiles* содержит базу данных в файлах с отдельными подкаталогами для сериализованных файлов ошибок и комментариев.
- *DbaseShelve* содержит основанную на shelve базу данных с отдельными файлами для сообщений об ошибках и комментариев.

С содержимым подкаталогов баз данных мы познакомимся позже в этой главе при изучении реализации базы данных.

Стратегия представления

PyErrata доводит разделение на подзадачи, повторное использование кода и инкапсуляцию до крайней степени. Например, сценарии верхнего уровня часто состоят всего из нескольких строк и в конечном счете вызывают общие логические части в совместно используемых вспомогательных модулях. При такой архитектуре короткие сегменты кода вперемешку с массой снимков экранов затрудняют прослеживание логики выполнения программы.

Чтобы облегчить изучение системы, мы примем здесь несколько иной подход. Реализация PyErrata будет представлена тремя главными разделами, соответствующими основным областям функционирования системы: просмотру сообщений, подаче сообщений и интерфейсам базы данных. Перед этими тремя разделами будет показана корневая страница, но в основном для контекста – это простой статический HTML.

В разделах, посвященных просмотру и передаче, сначала показываются все модели взаимодействия с пользователем (и снимки экранов), а затем следует исходный код, реализующий это взаимодействие. Как и пример PyForm главы 16 «Базы данных и постоянное хранение», PyErrata в глубине своей является программой доступа к базе данных, и ее интерфейсы к базе данных в конечном счете составляют ядро системы. Однако поскольку в этих интерфейсах инкапсулируется большинство деталей, относящихся к нижнему уровню хранения, их представление будет оставлено напоследок.

Хотя какие-то перемещения за функциональные границы, чтобы найти некоторые модули, все же могут потребоваться, такая структура размещения кода главных частей системы по отдельным разделам должна свести к минимуму необходимость листать страницы.

Смотри исходный код

Хочу сделать здесь стандартное предупреждение для изучения конкретных примеров: хотя по ходу этой главы разъясняются главные концепции, разбираться со всем в целом вам придется, в известной мере, самостоятельно. Как всегда, обращайтесь к листингам исходного кода в этой главе (и на CD), если о каких-то деталях не сказано явно. Я умышленно выбрал такой минималистский подход главным образом потому, что предполагаю наличие у вас достаточных знаний о CGI-программировании и языке Python, если вы добрались до этого места в книге, но также и потому, что на практике во время разработки чтение чужого кода занимает столько же времени, сколько написание своего. Python делает обе эти задачи относительно легкими, но теперь вам предоставляется возможность убедиться в этом самостоятельно.

Я хочу сразу признаться, что у этой главы есть скрытый план. PyErrata не только демонстрирует новые приемы создания сценариев для серверов, но и иллюстрирует стандартные концепции разработки на Python в целом. Попутно мы обратим внимание на текущую архитектуру программного обеспечения этой системы и отметим ряд альтернативных проектных решений. Обязательно обратите внимание на распределение логики по нескольким уровням абстракции. Например, в результате разделения кода базы данных и интерфейса пользователя (генерации страниц) сокращаются избыточность кода и зависимости между модулями и максимизируется повторное использование кода. Такая технология полезна во всех системах Python, не только тех, которые относятся к веб.

Корневая страница

Начнем с самого начала. В этой главе будет целиком изучена реализация PyErrata, но читателям рекомендуется также посетить сайт, где она находится, чтобы своими руками почувствовать взаимодействие с ней. В отличие от PyMailCgi, доступ к PyErrata не защищается паролем, поэтому можно обращаться ко всем ее страницам без какой-либо дополнительной настройки.

PyErrata состоит из группы файлов HTML и сценариев Python CGI, а также нескольких графических файлов. Как обычно, при изучении этой главы можно просто направить свой веб-броузер на корневую страницу системы, чтобы запустить ее живьем. Корневая страница в настоящее время находится здесь:[1]

http://starship.python.net/~lutz/PyErrata/pyerrata.html

Если пойти по этому адресу, то броузеру будет выдана страница, показанная на рис. 14.2. PyErrata поддерживает как передачу, так и просмотр комментариев и сообщений об ошибках; четыре основные ссылки на этой странице фактически предоставляют доступ к ее базам данных по записи и по чтению через веб.

Файл со статическим кодом HTML, при загрузке которого отображается эта страница, приведен в примере 14.1. То что нас интересует в нем, выделено полужирным

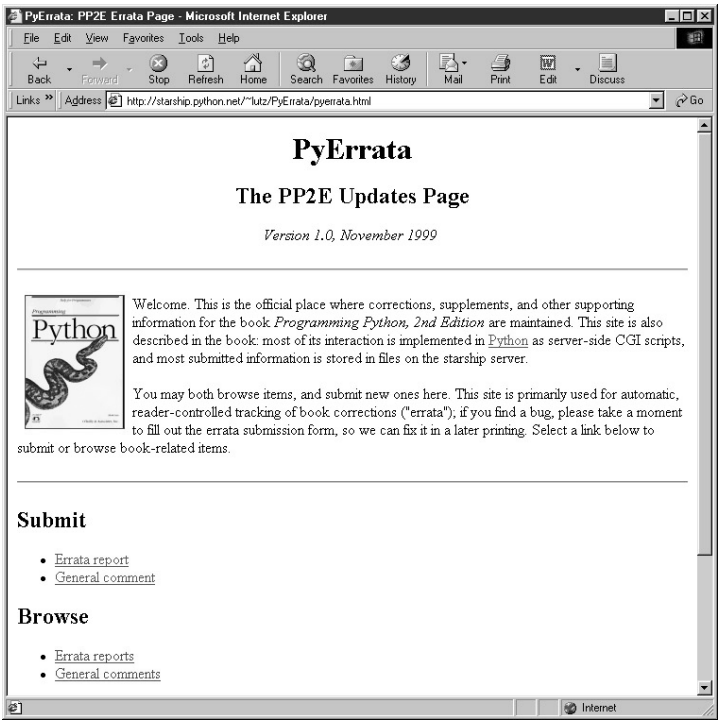


Рис. 14.2. Главная страница PyErrata

[1] Если к тому времени, когда вы читаете эту книгу, ссылка окажется недействующей, найдите новую ссылку на сайте этой книги *http://rmi.net/~lutz/about-pp.html*. Похоже, что сайты меняют адреса быстрее, чем программисты место работы.

шрифтом: ссылки на страницы передачи и просмотра для комментариев и ошибок. На этой странице есть другие ссылки, но мы займемся только частями, показанными на снимке экрана. Например, на сайте в конечном счете будут также размещены файлы HTML-страниц ресурсов (например, ресурсы и изменения в Python), но мы не рассматриваем их в этой книге.

Пример 14.1. PP2E\Internet\Cgi-Web\PyErrata\pyerrata.html

```
<HTML><BODY>
<TITLE>PyErrata: PP2E Errata Page</TITLE>
<H1 align=center>PyErrata</H1>
<H2 align=center>The PP2E Updates Page</H2>
<P  align=center><I>Version 1.0, November 1999</I></P>

<HR><P>
<A href="http://rmi.net/~lutz/about-pp.html">
<IMG src="ppsmall.gif" align=left alt="[Book Cover]" border=1 hspace=8></A>

Welcome.  This is the official place where corrections, supplements,
and other supporting information for the book <I>Programming Python,
2nd Edition</I> are maintained.  This site is also described in the book:
most of its interaction is implemented in
<A HREF="http://rmi.net/~lutz/about-python.html">Python</A> as server-side
CGI scripts, and most submitted information is stored in files on the starship
server.
<P>
You may both browse items, and submit new ones here.  This site is primarily
used for automatic, reader-controlled tracking of book corrections ("errata");
if you find a bug, please take a moment to fill out the errata submission
form, so we can fix it in a later printing.  Select a link below to submit
or browse book-related items.
</P>
<HR>

<H2>Submit</H2>
<UL>
<LI><A href="submitErrata.html">Errata report</A>
<LI><A href="submitComment.html">General comment</A>
</UL>

<H2>Browse</H2>
<UL>
<LI><A href="browseErrata.html">Errata reports</A>
<LI><A href="browseComments.html">General comments</A>
</UL>

<H2>Library</H2>
<UL>
<LI><A href="resourceSupplements.html">Supplements</A>
<LI><A href="resourcePythonchanges.html">Python changes</A>
<LI><A href="resourcePatchfiles.html">Program patch files</A>
</UL>

<HR>
<A href="http://www.python.org">
<IMG SRC="PythonPoweredSmall.gif"
 ALIGN=left ALT="[Python Logo]" border=0 hspace=10></A>
<A href="../PyInternetDemos.html">More examples</A>
</BODY></HTML>
```

Просмотр сообщений PyErrata

Перейдем к первой важной функции системы: просмотру записей с сообщениями. Прежде чем изучать код, осуществляющий операции просмотра, разберемся с тем, какого рода взаимодействия с пользователем он должен осуществлять. Если вам больше нравится сразу переходить к коду, можете сейчас пропустить два раздела, но не забудьте при изучении кода вернуться в это место, чтобы посмотреть на снимки экранов.

Интерфейс пользователя: просмотр сообщений с комментариями

Как показано на рис. 14.2, PyErrata позволяет просматривать и передавать два вида сообщений: общие комментарии и доклады об опечатках (программных ошибках). При щелчке по ссылке «General comments» в разделе Browse корневой страницы выводится страница, показанная на рис. 14.3.

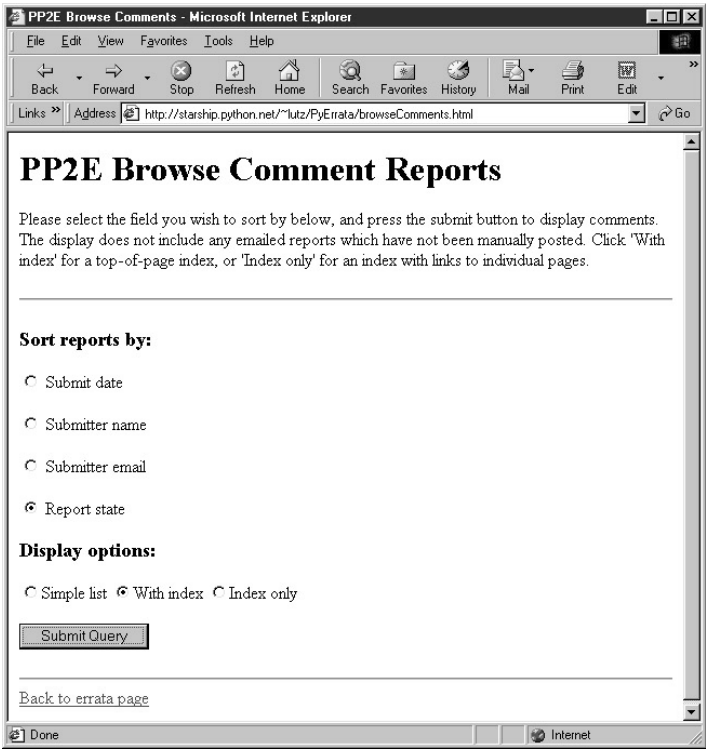


Рис. 14.3. Просмотр комментариев, страница выбора

Прежде всего, функция просмотра в PyErrata предоставляет пользователям возможность опрашивать и вводить сообщения в базу данных несколькими способами. Сообщения могут быть упорядочены по любому полю и выведены в трех разных форматах. Страницы просмотра верхнего уровня, по существу, служат тому, чтобы настраивать запрос к базе данных сообщений и представлять его результаты.

Чтобы задать способ упорядочения, выберите сначала критерий сортировки: имя поля сообщения, по которому будут упорядочены листинги сообщений. Поля представ-

лены на этой странице в виде переключателей. Чтобы задать формат вывода сообщений, выберите один из следующих:

- *Simple list* выбирает простую страницу с отсортированным списком.
- *With index* создает страницу с отсортированным списком и гиперссылками в верхней части, при щелчке по которым происходит переход в начальную точку каждого значения ключа сортировки.
- *Index only* создает страницу, содержащую только гиперссылки для каждого значения ключа сортировки, щелчок по которым загружает и выводит соответствующие записи.

Рис. 14.4 показывает простой случай, возникающий при щелчке по кнопке сортировки «Submit date» (дата передачи), выборе формата вывода «Simple list» и нажатии кнопки Submit Query для связи со сценарием Python на сервере. Это прокручиваемый список всех сообщений в базе данных с комментариями, упорядоченный по дате передачи.

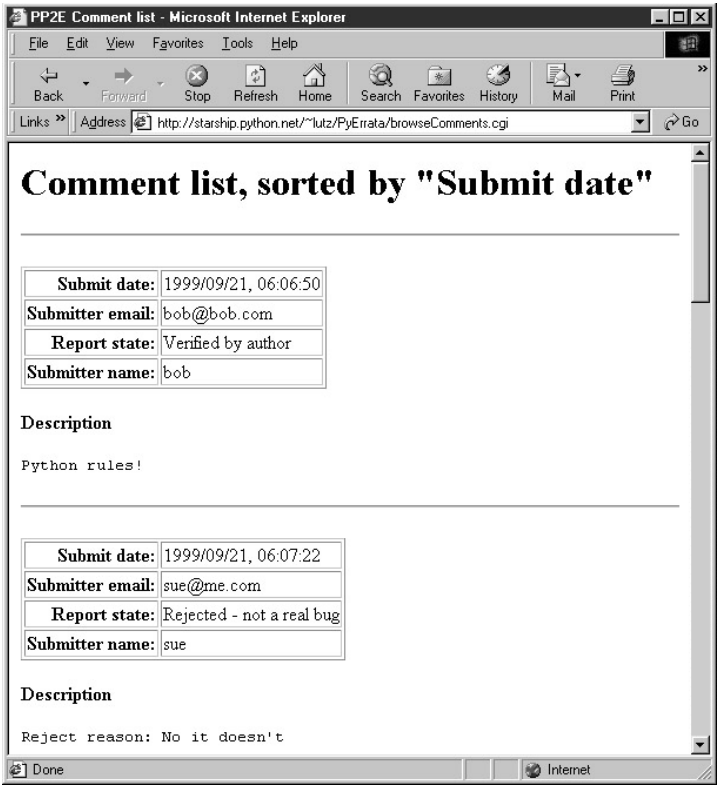


Рис. 14.4. Просмотр комментариев, вариант «Simple list»

Во всех результатах запросов каждая запись выводится в виде таблицы значений полей атрибутов (в количестве присутствующих в записи), за которой следует текст поля описания в записи. Описание обычно состоит из нескольких строк, поэтому оно показывается отдельно без форматирования HTML (то есть в том виде, как оно было введено). Если в списке есть несколько записей, они отделяются одна от другой горизонтальными линиями.

Такие простые списки хороши для небольших баз данных, а два других варианта вывода лучше подходят для больших групп сообщений. Например, если выбрать вариант «With index», будет получена страница, в начале которой находится список ссылок на другие точки на той же странице, за которым следует список записей, упорядоченных и сгруппированных по значению ключа сортировки. На рис. 14.5 показано использование варианта «With index» с ключом сортировки «Report state».

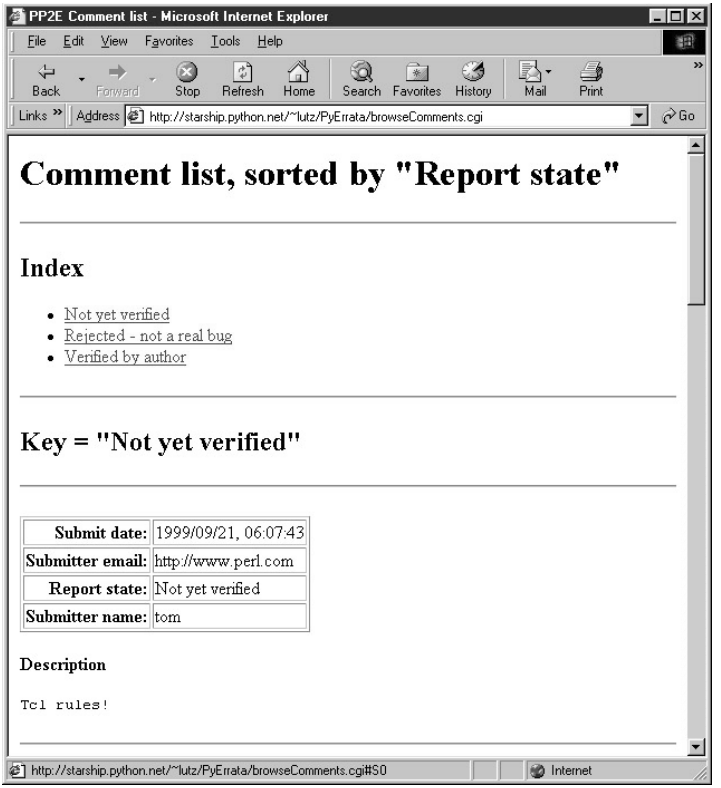


Рис. 14.5. Просмотр комментариев, вариант «With index»

Для просмотра сообщений пользователь может прокрутить список или щелкнуть по одной из ссылок вверху; они ведут к разделам списка сообщений, где начинаются записи с заданным значением ключа. Эти гиперссылки используют синтаксис ссылок на разделы вида *file.html#section*, поддерживаемый большинством броузеров, и внутристраничные теги. Важные части генерируемого кода HTML выглядят так:

```
<title>PP2E Comment list</title>
<h1>Comment list, sorted by "Report state"</h1><hr>
<h2>Index</h2><ul>
<li><a href="#S0">Not yet verified</a>
<li><a href="#S1">Rejected - not a real bug</a>
<li><a href="#S2">Verified by author</a>
</ul><hr>
<h2><a name="S0">Key = "Not yet verified"</a></h2><hr>
<p><table border>
<tr><th align=right>Submit date:<td>1999/09/21, 06:07:43
...еще...
```

Рис. 14.6 показывает результат щелчка по одной такой ссылке на странице, отсортированной по дате передачи. Обратите внимание на #S4 в конце URL результата. Чуть ниже мы увидим, как автоматически генерируются эти теги.

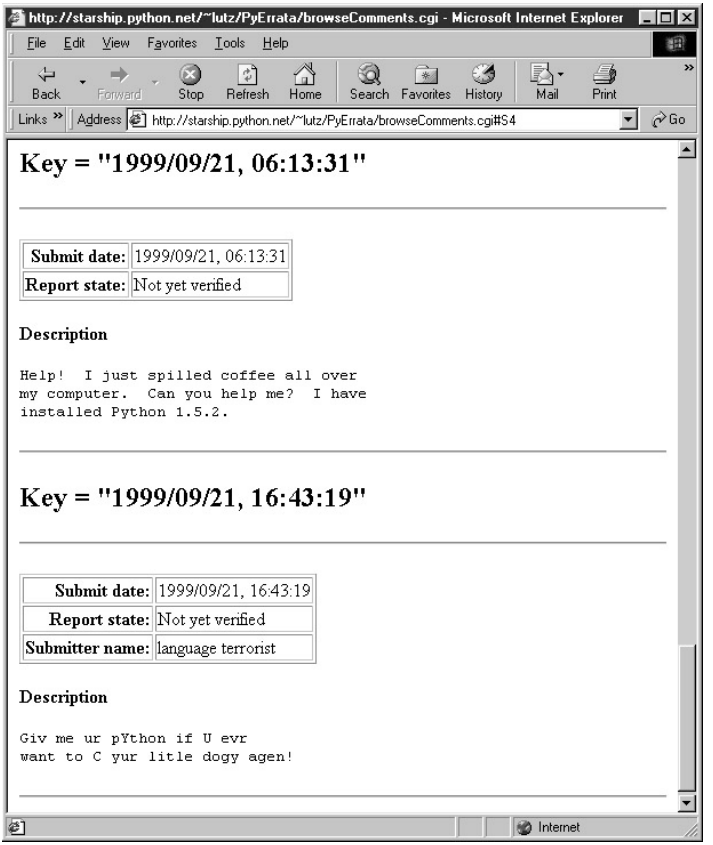


Рис. 14.6. Просмотр комментариев, вариант «With index»

Для очень больших баз данных вывод содержимого всех записей на одной странице может оказаться непрактичным; решение в этом случае дает третий вариант формата отображения PyErrata. На рис. 14.7 показана страница, созданная с параметром отображения «Index only» и упорядочением сообщений «Submit date». На этой странице нет записей, а есть только список гиперссылок, которые «умеют» получать записи с заданным значением ключа. Это еще один пример того, что мы окрестили *умными ссылками* – ключ и значение встроены в URL этих ссылок.

PyErrata генерирует эти ссылки динамически. Выглядят они следующим образом (за исключением переводов строки, добавленных для улучшения читаемости):

```
<title>PP2E Comment list</title>
<h1>Comment list, sorted by "Submit date"</h1><hr>
<h2>Index</h2><ul>
<li><a href="index.cgi?kind=Comment&
                      sortkey=Submit+date&
                      value=1999/09/21,+06%3a06%3a50">1999/09/21, 06:06:50</a>
<li><a href="index.cgi?kind=Comment&
```

```
                    sortkey=Submit+date&
                    value=1999/09/21,+06%3a07%3a22">1999/09/21, 06:07:22</a>
...еще...
</ul><hr>
```

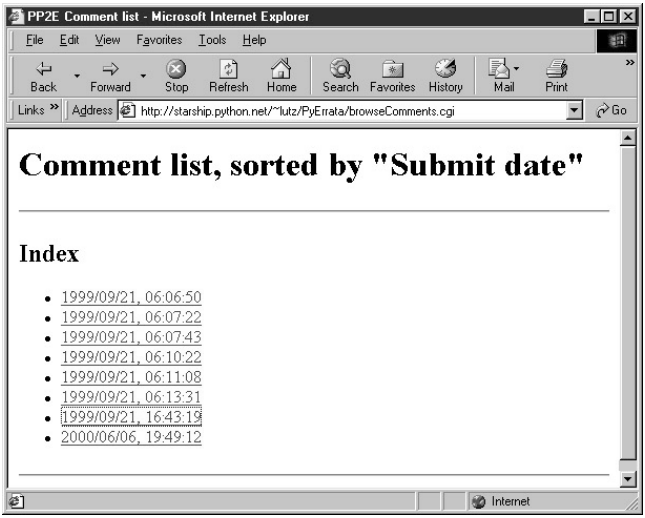


Рис. 14.7. Просмотр комментариев, список выбора «Index only»

Обратите внимание на этот раз на параметры в кодировке URL; как вы увидите в коде, это снова работа модуля Python `urllib`. Кроме того, обратите внимание, что в отличие от PyMailCgi – примера прошлой главы, PyErrata генерирует в списках минимальные URL (без имени сервера и пути – их угадывает по адресу предыдущей страницы и добавляет броузер). Если посмотреть на исходный код сгенерированной страницы, имеющиеся умные ссылки станут более очевидными; рис. 14.8 показывает код одной такой страницы указателя.[1]

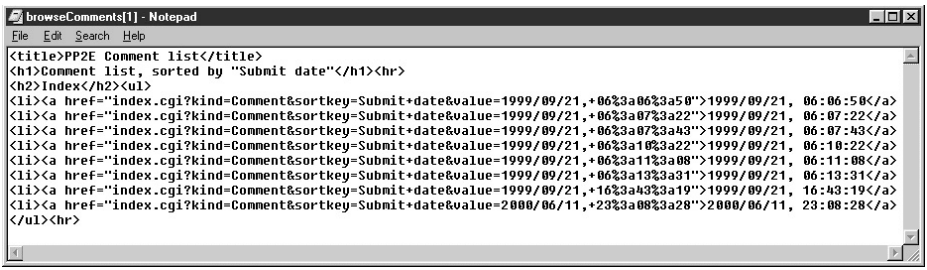


Рис. 14.8. Код ссылок, генерируемый PyErrata

При щелчке по ссылке на странице «Index only» загружаются и выводятся все записи базы данных с показанным значением в отображаемом поле ключа. Например, при нажатии предпоследней ссылки в странице указателя (рис. 14.7) выводится страни-

[1] Как и в PyMailCgi, символ `&` в генерируемых URL не кодируется PyErrata, так как имя его параметра не конфликтует с именами кодов символов HTML. Если ваши имена конфликтуют, примените `cgi.escape` к URL, вставляемым в страницы веб.

ца, показанная на рис. 14.9. Как обычно, сгенерированные ссылки появляются в поле адреса результата.

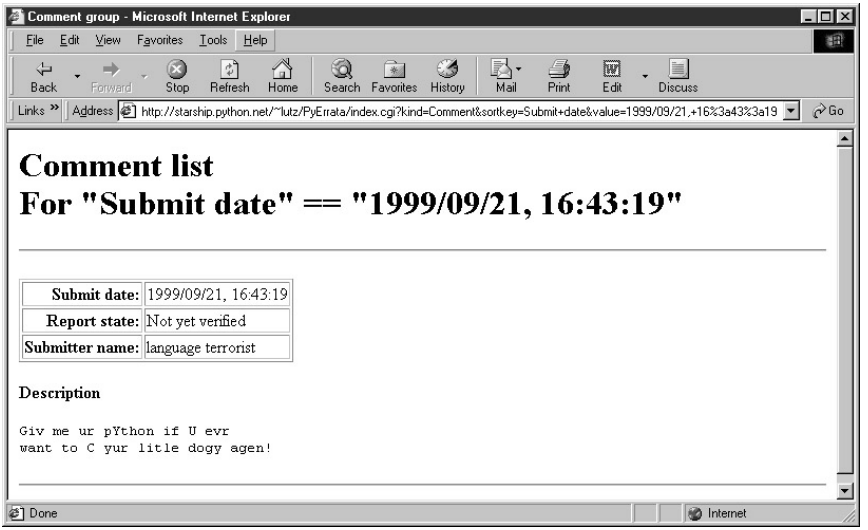


Рис. 14.9. Просмотр комментариев после щелчка по ссылке в «Index only»

Если запросить индекс по полю «Submitter name», будут сгенерированы аналогичные результаты, но с другими значениями ключа в списке и URL; рис. 14.10 показывает результат щелчка по ссылке на такой странице индекса. Это та же запись, что и на рис. 14.9, но доступ к ней был осуществлен по ключу имени, а не даты передачи. Благодаря обработке записей в общем виде PyErrata предоставляет несколько способов просмотра и доступа к хранящимся данным.

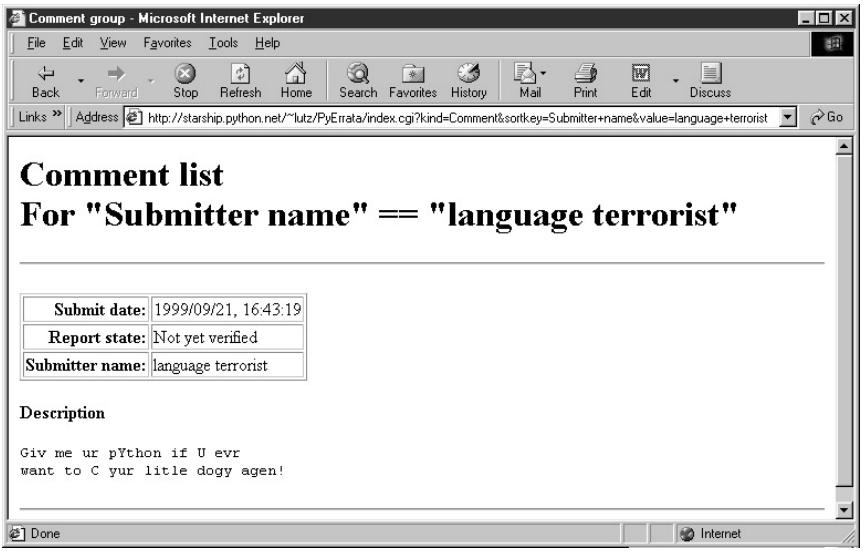


Рис. 14.10. Просмотр комментариев, страница «Index only»

Интерфейс пользователя: просмотр сообщений об ошибках

PyErrata поддерживает две разные базы данных – одну для общих комментариев, а другую для сообщений о настоящих ошибках. Для PyErrata записи являются просто объектами с полями; система одинаково обрабатывает комментарии и ошибки и готова воспользоваться любой базой данных, которая будет ей передана. Поэтому интерфейс для просмотра записей с ошибками почти идентичен интерфейсу для просмотра комментариев, и, как будет показано в разделе реализации, в основном использует тот же код.

Однако сообщения об ошибках содержат другие поля. Поскольку полей значительно больше, чем здесь можно заполнить, корневая страница функции просмотра сообщений об ошибках несколько отличается. Как видно из рис. 14.11, поля сортировки выбираются из выпадающего списка, а не с помощью переключателей. В качестве ключа сортировки могут использоваться любые атрибуты сообщения об ошибке, даже если для выбранного поля в некоторых сообщениях нет значения. Большинство полей не являются обязательными. Как мы увидим ниже, сообщения с пустыми значениями полей показываются со значением ? в индексных списках и группируются под значением (none) в списках сообщений.

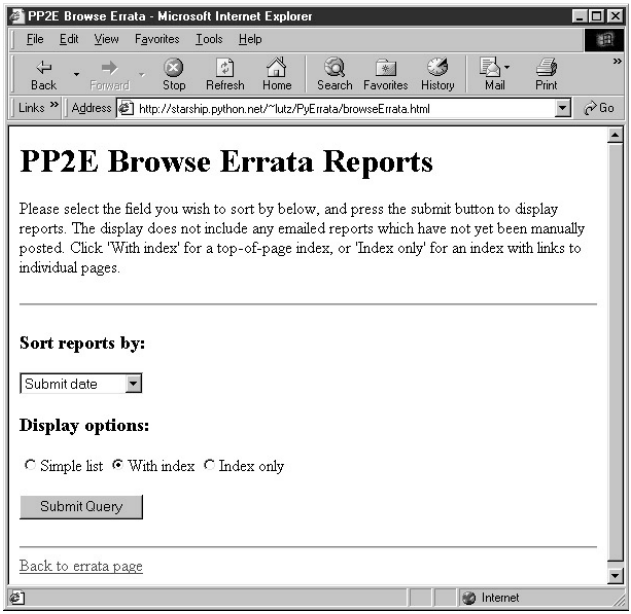


Рис. 14.11. Просмотр сообщений об ошибках, страница выбора

После выбора порядка сортировки и формата вывода и передачи запроса все выглядит примерно так же, как для комментариев (хотя на метках пишется Errata, а не Comment). Например, рис. 14.12 показывает вариант «With index» для ошибок, отсортированных по дате передачи.

При щелчке по одной из ссылок на этой странице осуществляется переход к одному из разделов страницы сообщений об ошибках, как на рис. 14.13; и снова URL вверху использует гиперссылки `#section`.

Режим «Index only» здесь действует таким же образом: рис. 14.14 показывает индексную страницу для сортировки по полю «Chapter number». Обратите внимание на за-

пись «?»: при щелчке по ней будут получены и отображены все записи с пустым полем номера главы. При показе их пустые значения ключа выводятся как (none). В базе данных они действительно являются пустыми строками.

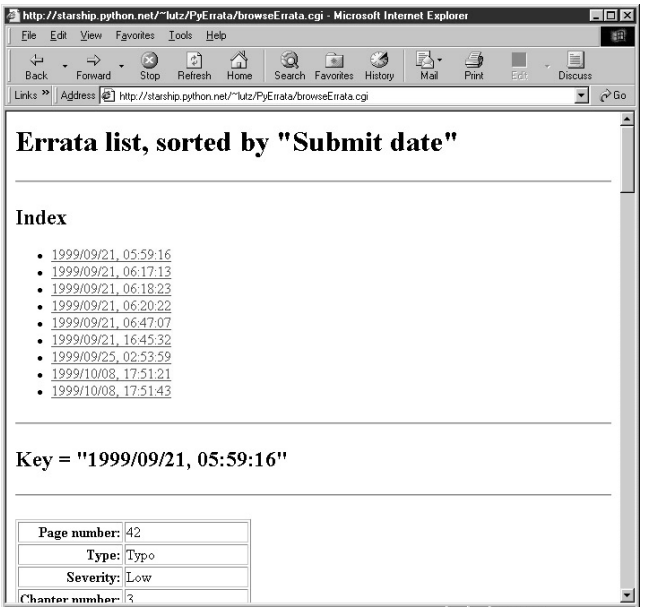


Рис. 14.12. Просмотр сообщений об ошибках, страница «With index»

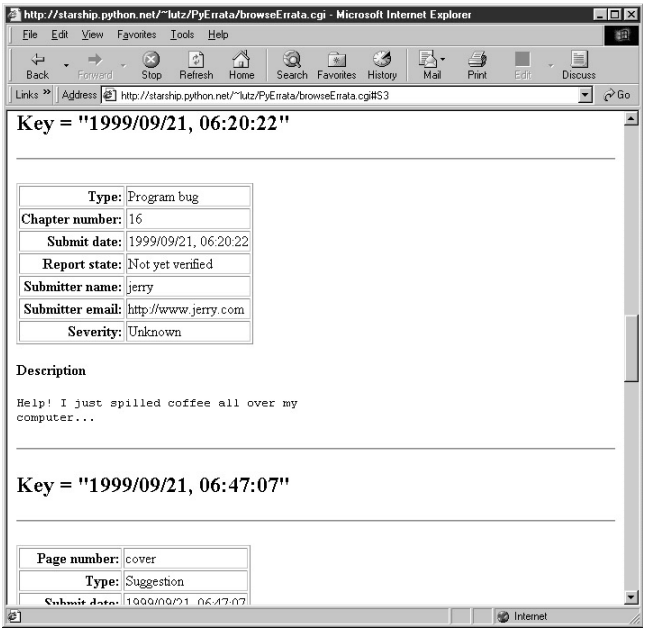


Рис. 14.13. Просмотр сообщений об ошибках, список сообщений

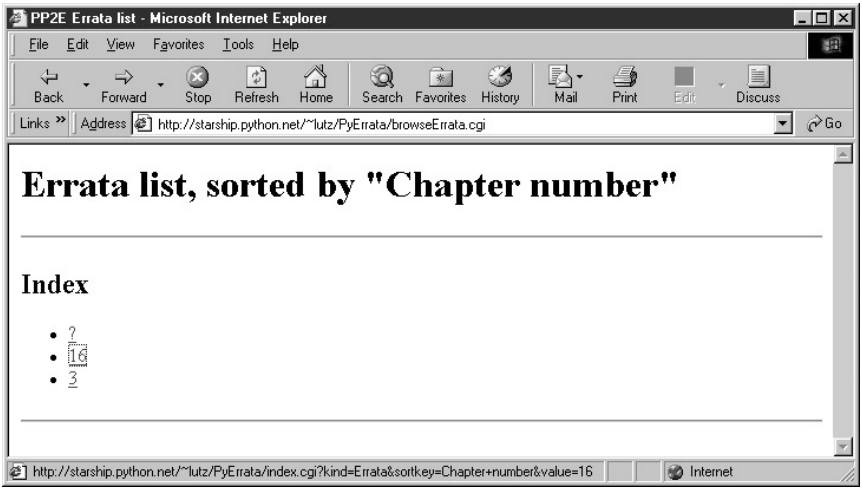


Рис. 14.14. Просмотр сообщений об ошибках, страница ссылок «Index only»

Щелчок по записи «16» выводит все сообщения об ошибках, помеченных в базе данных этим номером главы; рис. 14.15 показывает, что на этот раз было найдено только одно такое сообщение.

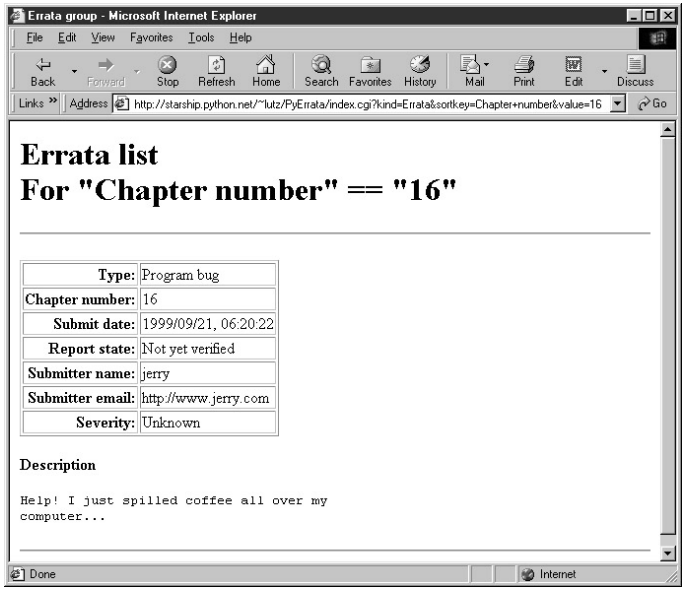


Рис. 14.15. Просмотр сообщений об ошибках, щелкнута ссылка «Index only»

Использование в PyErrata явных URL

Поскольку модуль Python `cgi` обрабатывает входные данные форм и параметры URL одинаковым образом, для генерации большинства уже показанных страниц можно

также использовать явные URL. На самом деле в PyErrata так и делается; показанный в верхней части рис. 14.15 URL:

```
http://starship.python.net/~lutz/
      PyErrata/index.cgi?kind=Errata&sortkey=Chapter+number&value=16
```

был внутренне сгенерирован PyErrata, представляя запрос, который должен быть отправлен следующему сценарию (большую часть его, с начала и до *PyErrata/*, фактически добавляет броузер). Но ничто не мешает пользователю (или другому сценарию) передать этот полностью заданный URL явно, инициируя запрос и ответ. Другие страницы тоже можно получить с помощью прямых URL; вот, например, URL для загрузки самой индексной страницы:

```
http://starship.python.net/~lutz/
      PyErrata/browseErrata.cgi?key=Chapter+number&display=indexonly
```

Аналогично, если вы хотите узнать у системы обо всех комментариях, переданных под некоторым именем, можно перемещаться по страницам запросов системы или ввести такой URL:

```
http://starship.python.net/~lutz/
      PyErrata/index.cgi?kind=Comment&sortkey=Submitter+name&value=Bob
```

Если в указанной базе данных нет соответствия с заданным значением ключа, будет получена страница с информацией об исключительной ситуации Python. Если же нужно просто получить список комментариев, отсортированный по дате передачи (например, для анализа в другом сценарии), введите такой URL:

```
http://starship.python.net/~lutz/
      PyErrata/browseComments.cgi?key=Submit+date&display=list
```

При обращениях такого рода к этой системе не со страниц ее форм необходимо задавать полный URL и значения параметров в кодировке URL. Памяти о предыдущей странице нет, а поскольку большинство значений ключей образуется из значений в сообщениях, переданных пользователями, в них могут содержаться произвольные данные.

Можно также использовать явные URL для передачи новых сообщений – каждое поле можно передать как параметр URL для сценария передачи:

```
http://starship.python.net/~lutz/
      PyErrata/submitComment.cgi?Description=spam&Submitter+name=Bob
```

Мы не поймем до конца, что при этом происходит, пока не доберемся до раздела «Передача сообщений в PyErrata».

Реализация: просмотр сообщений с комментариями

Теперь, посмотрев на внешнее поведение функции просмотра, засучим рукава и приступим к изучению ее реализации. В следующих разделах приведены и обсуждаются файлы исходного кода, реализующие в PyErrata операции просмотра. Все они располагаются на веб-сервере; одни являются статическими файлами HTML, а другие – исполняемыми сценариями Python. При чтении не забывайте обращаться к предыдущим разделам, посвященным интерфейсу пользователя, чтобы посмотреть, какие страницы создаются изучаемым кодом.

Как уже отмечалось выше, эта система разложена на части для повторного использования кода: сценарии верхнего уровня в основном лишь обращаются к обобщенным модулям, передавая им надлежащие параметры. База данных, где хранятся переданные со-

общения, тоже полностью инкапсулирована; ее реализацию мы изучим далее в этой главе, а пока можно в целом не обращать внимание на способ хранения информации.

Файл примера 14.2 реализует страницу верхнего уровня для просмотра комментариев.

Пример 14.2. PP2E\Internet\Cgi-Web\PyErrata\browseComments.html

```
<html><body bgcolor="#FFFFFF">
<title>PP2E Browse Comments</title>
<h1>PP2E Browse Comment Reports</h1>

<p>Please select the field  you wish to sort by below, and press
the submit button to display comments.  The display does not include
any emailed reports which have not been manually posted.  Click
'With index' for a top-of-page index, or 'Index only' for an index
with links to individual pages.
</p>

<hr>
<form method=POST action="browseComments.cgi">
      <h3>Sort reports by:</h3>

      <p><input type=radio name=key value="Submit date" checked> Submit date
      <p><input type=radio name=key value="Submitter name"> Submitter name
      <p><input type=radio name=key value="Submitter email"> Submitter email
      <p><input type=radio name=key value="Report state"> Report state

      <h3>Display options:</h3>
       <p><input type=radio name=display value="list">Simple list
          <input type=radio name=display value="indexed" checked>With index
          <input type=radio name=display value="indexonly">Index only
      <p><input type=submit>
</form>

<hr>
<a href="pyerrata.html">Back to errata page</A>
</body></html>
```

Это простой статический код HTML, в отличие от сценария (здесь нечего строить динамически). Как и на всех формах, щелчок по кнопке передачи запускает на сервере сценарий CGI (пример 14.3), которому передаются значения всех полей ввода.

Пример 14.3. PP2E\Internet\Cgi-Web\PyErrata\browseComments.cgi

```
#!/usr/bin/python

from dbswitch import DbaseComment        # dbfiles или dbshelve
from browse   import generatePage        # форматер html
generatePage(DbaseComment, 'Comment')    # загрузка данных, отправка страницы
```

Здесь немного действий, поскольку вся механика выполнения запроса выделена в модуль `browse` (показанный в примере 14.6), чтобы ее можно было использовать также для просмотра сообщений об ошибках. Внутренний механизм просмотра обоих типов записей одинаков; здесь мы передаем в качестве параметров только те элементы, которые различны в операциях просмотра комментариев и ошибок, а именно объект базы данных комментариев и метку «Comment» для создаваемых страниц. Модуль `browse` готов запросить и вывести записи из любой базы данных, которая ему передана.

Используемый здесь модуль `dbswitch` (приведенный в примере 14.13) просто осуществляет выбор между механизмами плоского файла и базы данных shelve. В результате помещения механизма выбора в один модуль потребуется обновить только один файл,

чтобы перейти на другой способ хранения; этот сценарий CGI совершенно не зависит от лежащего в его основе механизма базы данных. Технически объект `dbswitch.DbaseComment` является объектом *класса*, с помощью которого в дальнейшем строится объект интерфейса к базе данных в модуле `browse`.

Реализация: просмотр сообщений об ошибках

Файл примера 14.4 реализует страницу верхнего уровня просмотра сообщений об ошибках, с помощью которой устанавливаются порядок сортировки и формат вывода. На этот раз поля находятся в выпадающем списке выбора, но в остальном эта страница аналогична той, которая используется для комментариев.

Пример 14.4. PP2E\Internet\Cgi-Web\PyErrata\browseErrata.html

```
<html><body bgcolor="#FFFFFF">
<title>PP2E Browse Errata</title>
<h1>PP2E Browse Errata Reports</h1>

<p>Please select the field  you wish to sort by below, and press
the submit button to display reports.  The display does not include
any emailed reports which have not yet been manually posted.  Click
'With index' for a top-of-page index, or 'Index only' for an index
with links to individual pages.
</p>

<hr>
<form method=POST action="browseErrata.cgi">
      <h3>Sort reports by:</h3>
      <select name=key>
          <option>Page number
          <option>Type
          <option>Submit date
          <option>Severity
          <option>Chapter number
          <option>Part number
          <option>Printing date
          <option>Submitter name
          <option>Submitter email
          <option>Report state
      </select>
      <h3>Display options:</h3>
       <p><input type=radio name=display value="list">Simple list
          <input type=radio name=display value="indexed" checked>With index
          <input type=radio name=display value="indexonly">Index only
      <p><input type=submit>
</form>

<hr>
<a href="pyerrata.html">Back to errata page</A>
</body></html>
```

При передаче формы этого файла HTML на сервере вызывается сценарий примера 14.5.

Пример 14.5. PP2E\Internet\Cgi-Web\PyErrata\browseErrata.cgi

```
#!/usr/bin/python

from dbswitch import DbaseErrata         # dbfiles или dbshelve
from browse   import generatePage        # форматер html
generatePage(DbaseErrata)                # загрузка данных, отправка страницы
```

Опять-таки, здесь мало тем для разговора. В действительности этот сценарий почти совпадает со сценарием для просмотра комментариев, потому что в обоих используется логика, выделенная в модуль `browse`. Мы только передаем здесь для обработки другую базу данных.

Общие вспомогательные модули для просмотра

Для полного понимания операций просмотра необходимо изучить модуль примера 14.6, используемый в операциях просмотра как комментариев, так и ошибок.

Пример 14.6. PP2E\Internet\Cgi-Web\PyErrata\browse.py

```
########################################################################################################
# По запросу просмотра: получить и вывести данные на новой странице; данные сообщений
# хранятся в словарях в базе данных; предупреждение: ссылки разделов '#Si', генерируемые
# для индекса в начале страницы, работают в новом Internet Explorer, но могут отказать
# в старых Netscape; в случае отказа попробуйте режим 'index only', в котором данные
# для создания новой страницы кодируются в ссылках  url; ссылки url должны кодироваться
# с помощью urllib, а не cgi.escape (для текста, помещаемого в ответный поток html;
# IE автоматически заменяет пробелы на %20 при щелчке по url, поэтому замена '+'
# не всегда требуется, но urllib.quote_plus более надежный; веб-броузер добавляет
# http://server-name/root-dir/PyErrata/ к indexurl;
########################################################################################################
import cgi, urllib, sys, string
sys.stderr = sys.stdout             # показывать ошибки в броузере
indexurl = 'index.cgi'              # минимальные url в ссылках

def generateRecord(record):
    print '<p><table border>'
    rowhtml = '<tr><th align=right>%s:<td>%s\n'
    for field in record.keys():
        if record[field] != '' and field != 'Description':
            print rowhtml % (field, cgi.escape(str(record[field])))

    print '</table></p>'
    field = 'Description'
    text  = string.strip(record[field])
    print '<p><b>%s</b><br><pre>%s</pre><hr>' % (field, cgi.escape(text))

def generateSimpleList(dbase, sortkey):
    records = dbase().loadSortedTable(sortkey)           # создать список
    for record in records:
        generateRecord(record)

def generateIndexOnly(dbase, sortkey, kind):
    keys, index = dbase().loadIndexedTable(sortkey)      # создать ссылки индекса
    print '<h2>Index</h2><ul>'                           # для загрузки при щелчке
    for key in keys:
        html = '<li><a href="%s?kind=%s&sortkey=%s&value=%s">%s</a>'
        htmlkey    = cgi.escape(str(key))
        urlkey     = urllib.quote_plus(str(key))         # кодировка html или url
        urlsortkey = urllib.quote_plus(sortkey)          # заменить пробелы на '+'
        print html % (indexurl,
                      kind, urlsortkey, (urlkey or '(none)'), (htmlkey or '?'))
    print '</ul><hr>'

def generateIndexed(dbase, sortkey):
    keys, index = dbase().loadIndexedTable(sortkey)
    print '<h2>Index</h2><ul>'
    section = 0                                          # создать указатель
```

```
        for key in keys:
            html = '<li><a href="#S%d">%s</a>'
            print html % (section, cgi.escape(str(key)) or '?')
            section = section + 1
        print '</ul><hr>'
        section = 0                                          # создать детали
        for key in keys:
            html = '<h2><a name="#S%d">Key = "%s"</a></h2><hr>'
            print html % (section, cgi.escape(str(key)))
            for record in index[key]:
                generateRecord(record)
            section = section + 1

    def generatePage(dbase, kind='Errata'):
        form = cgi.FieldStorage()
        try:
            sortkey = form['key'].value
        except KeyError:
            sortkey = None

        print 'Content-type: text/html\n'
        print '<title>PP2E %s list</title>' % kind
        print '<h1>%s list, sorted by "%s"</h1><hr>' % (kind, str(sortkey))

        if not form.has_key('display'):
            generateSimpleList(dbase, sortkey)

        elif form['display'].value == 'list':              # отправить согласно типу вывода
            generateSimpleList(dbase, sortkey)             # можно использовать здесь словарь

        elif form['display'].value == 'indexonly':
            generateIndexOnly(dbase, sortkey, kind)

        elif form['display'].value == 'indexed':
            generateIndexed(dbase, sortkey)
```

Этот модуль, в свою очередь, сильно зависит от интерфейсов баз данных верхнего уровня, с которыми мы вскоре познакомимся. А пока на данном верхнем уровне абстракции нам требуется лишь знать, что база данных экспортирует *интерфейсы* для загрузки записей с сообщениями и сортировки и группировки их по значениям ключа и что записи с сообщениями хранятся в базе данных в виде *словарей* с одним ключом для каждого поля в сообщении. Есть два интерфейса верхнего уровня для доступа к хранящимся сообщениям:

- `dbase().loadSortedTable(sortkey)` загружает записи из создаваемого объекта интерфейса базы данных в простой список, отсортированный по ключу, имя которого передается в качестве параметра. Он возвращает список словарей записей, отсортированный по полю записи.
- `dbase().loadIndexedTable(sortkey)` загружает записи из создаваемого объекта интерфейса базы данных в словарь списков, сгруппированных по значениям переданного ключа (одна запись словаря на каждое значение ключа сортировки). Он возвращает как словарь списков словарей записей, представляющий группировку по ключу, так и список отсортированных ключей для упорядоченного доступа к словарю групп (вспомните, что словари не упорядочены).

В варианте вывода простого списка используется первый вызов, а в обоих вариантах вывода индекса используется второй вызов для построения списков значений ключей и групп соответствующих им записей. Мы увидим реализацию этих вызовов и вызо-

вов сохранения записей позднее. Сейчас нас интересует только, чтобы они работали, как заявлено.

Технически говоря, можно использовать любое отображение для сохранения полей записи сообщения в базе данных, но в данной версии системы в качестве единицы хранения используются словари. Для выбора этого представления есть достаточные основания:

- Оно хорошо сочетается с используемым в CGI объектом входных полей формы, который возвращает `cgi.FieldStorage`. Сценарии передачи сообщений просто соединяют словари входных полей формы со словарями ожидаемых полей, чтобы сформировать запись.
- Это представление более непосредственно, чем другие. Например, можно легко обработать все поля общим образом, перебрав список ключей словаря записи, в то время как использование классов и имен атрибутов для полей является менее прямым и может потребовать частых вызовов `getattr`.
- Такое представление является более гибким, чем другие. Например, ключи словарей могут принимать значения, недопустимые в именах атрибутов (скажем, с пробелами внутри), и потому хорошо отображаются на произвольные имена полей формы.

Подробнее о базах данных будет сказано ниже. В режиме вывода «Index only» модуль `browse` генерирует ссылки, щелчок по которым запускает сценарий примера 14.7. В этом файле тоже мало интересного, потому что генерация страницы снова по большей части возложена на функцию `generateRecord` в модуле `browse` из примера 14.6. Передаваемое ей поле «kind» используется для выбора надлежащего класса объекта базы данных, которому передается запрос; переданное имя поля сортировки и значения ключей используются затем для извлечения соответствующих записей, возвращаемых интерфейсом базы данных.

Пример 14.7. PP2E\Internet\Cgi-Web\PyErrata\index.cgi

```
#!/usr/bin/python
###############################################################################################
# Выполняется при щелчке по гиперссылке, генерируемой browse.py для режима "index-only";
# входные параметры жестко закодированы в url ссылки, но ничто не мешает кому-либо
# самостоятельно создать аналогичную ссылку--не применяйте eval() к входным данным
# (из соображений безопасности); учтите, что в этом сценарии предполагается, что после создания
# страницы индекса не удалялись файлы с данными; cgi.FieldStorage обращает кодировку urllib
# во входных параметрах (%xx и '+' заменяются пробелами);
###############################################################################################

import cgi, sys, dbswitch
from browse import generateRecord
sys.stderr = sys.stdout
aorm = cgi.FieldStorage()                                   # обращает кодировку url

inputs = {'kind':'?', 'sortkey':'?', 'value':'?'}
for field in inputs.keys():
    if form.has_key(field):
        inputs[field] = cgi.escape(form[field].value)   # добавляет кодировку html

if inputs['kind'] == 'Errata':
    dbase = dbswitch.DbaseErrata
else:
    dbase = dbswitch.DbaseComment

print 'Content-type: text/html\n'
print '<title>%s group</title>' % inputs['kind']
```

```
print '<h1>%(kind)s list<br>For "%(sortkey)s" == "%(value)s"</h1><hr>' % inputs

keys, index = dbase().loadIndexedTable(inputs['sortkey'])
key = inputs['value']
if key == '(none)': key = ''
for record in index[key]:
    generateRecord(record)
```

В некотором смысле этот сценарий `index` служит продолжением `browse` с выводом страницы в промежутке. Можно было бы объединить эти файлы ценой некоторого труда и увеличения сложности, но их логика действительно должна выполняться в отдельных *процессах*. В интерактивных программах на стороне клиента остановка для ввода данных пользователем могла бы просто иметь вид вызова функции (например, `raw_input`); в области CGI такая остановка обычно требует порождения отдельного процесса для осуществления ввода.

Есть еще два момента, которые нужно подчеркнуть, прежде чем двигаться дальше. Прежде всего, в варианте «With index» есть свои ограничения. Обратите внимание на то, как модуль `browse` генерирует внутристраничные гиперссылки `#section`, а затем помечает раздел для каждого ключа в списке записей заголовочной строкой с тегом `<A name=#section>`, используя счетчик для создания уникальных меток разделов. Все это основано на том факте, что интерфейс базы данных умеет возвращать записи, сгруппированные по значениям ключа (один список для каждого ключа). К сожалению, такие ссылки внутри одной страницы работают не во всех броузерах (они не действуют в старых Netscape); если у вас они не работают, для доступа к записям по группам ключей воспользуйтесь вариантом «Index only».

Второй момент связан с тем, что, поскольку все поля сообщения являются необязательными, система должна красиво обработать пустые или отсутствующие поля. Поскольку в сценариях передачи (описываемых в следующем разделе) для каждого типа записей определен фиксированный набор полей, в действительности в записях базы данных никогда нет «отсутствующих» полей; пустые поля просто хранятся в виде пустых строк и опускаются при показе записей. Когда пустые значения оказываются в индексных списках, они отображаются как `?`; в метках ключей и URL они обозначаются строкой `(none)`, которая внутренне отображается в пустую строку в только что приведенных модулях `index` и `browse` (пустые строки нехорошо использовать в качестве параметров URL). Это тонкая особенность, и за подробностями обратитесь к коду этих модулей.

Несколько слов об избыточности: обратите внимание на то, что список возможных полей сортировки на страницах ввода для просмотра жестко закодирован в их файлах HTML. Поскольку сценарии передачи, которые мы рассмотрим в следующем разделе, обеспечивают одинаковый набор полей у всех записей в базе данных, списки в HTML-файлах оказываются избыточными, когда в базах данных хранятся записи.

В принципе, можно было бы построить списки полей сортировки для HTML, посмотрев ключи любой записи в базах данных комментариев или ошибок (подобно тому, как мы делали это в примере с выбором языка в главе 12 «Сценарии на стороне сервера»), но для этого потребуются дополнительные операции с базой данных. Эти списки также частично пересекаются со списком полей в HTML страницы передачи и сценариях передачи, но все же достаточно различаются, чтобы оправдать некоторую избыточность.

Передача сообщений в PyErrata

Следующая важная функциональная область PyErrata служит для передачи создаваемых пользователем новых сообщений с комментариями или ошибками. Как и прежде, начнем с освоения модели интерфейса пользователя этой составной части, а потом изучим ее код.

Интерфейс пользователя: передача сообщений с комментариями

Как мы видели, PyErrata поддерживает две пользовательские функции: просмотр базы данных сообщений и добавление в нее новых сообщений. Если щелкнуть по ссылке «General comment» в секции Submit корневой страницы, показанной на рис. 14.2, то будет возвращена страница для передачи комментариев, показанная на рис. 14.16.

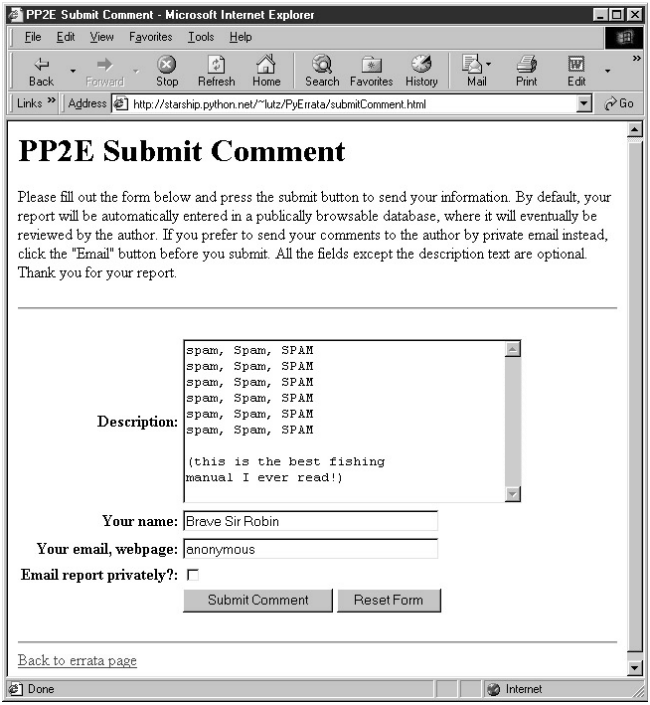


Рис. 14.16. Передача комментариев, страница ввода

Изначально эта страница показывается пустой. Данные, вводимые в поля формы, при нажатии внизу кнопки передачи посылаются сценарию на сервере. Если система смогла сохранить данные в новой записи базы данных, клиент получает страницу подтверждения, показанную на рис. 14.17.

Все поля формы передачи не обязательны, за исключением одного: если оставить пустым поле «Description» и послать форму, возвращается страница ошибки, показанная на рис. 14.18 (генерируется во время передачи сообщения об ошибке). Комментарии и сообщения об ошибках без описаний не слишком полезны, поэтому такие запросы выбрасываются. Все другие поля сообщения сохраняются как пустые, если при отправке сценариям передачи они не заполнены (или вообще отсутствуют).

Рис. 14.17. Передача комментариев, страница подтверждения

Рис. 14.18. Передача, страница ошибки при отсутствующем поле

После передачи комментария можно вернуться к страницам просмотра и увидеть сообщение в базе; на рис. 14.19 показано только что посланное сообщение, найденное по ключу «Submitter name», в формате вывода «With index».

Интерфейс пользователя: передача сообщений об ошибках

Здесь снова страницы, генерируемые для передачи сообщений об ошибках, практически совпадают с только что показанными для передачи комментариев, поскольку комментарии и ошибки обрабатываются в системе одинаково. Те и другие являются экземплярами общих записей базы данных с разными наборами полей. И так же, как прежде, страница передачи сообщений об ошибке верхнего уровня отличается, потому что там значительно больше полей, которые можно заполнять; на рис. 14.20 показана верхняя часть страницы ввода.

Здесь множество полей, но обязательным является только description. Мысль в том, чтобы пользователи заполняли для описания проблемы те поля, которые хотят; все текстовые поля по умолчанию принимают в качестве значения пустую строку, если в них ничего не введено. На рис. 14.21 показано сообщение, в котором большинство полей заполнено соответствующей информацией.

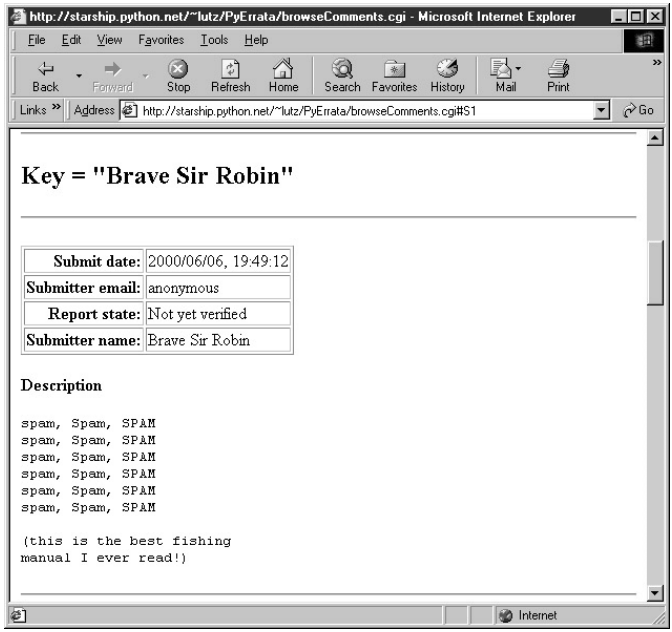


Рис. 14.19. Передача комментариев, проверка результата

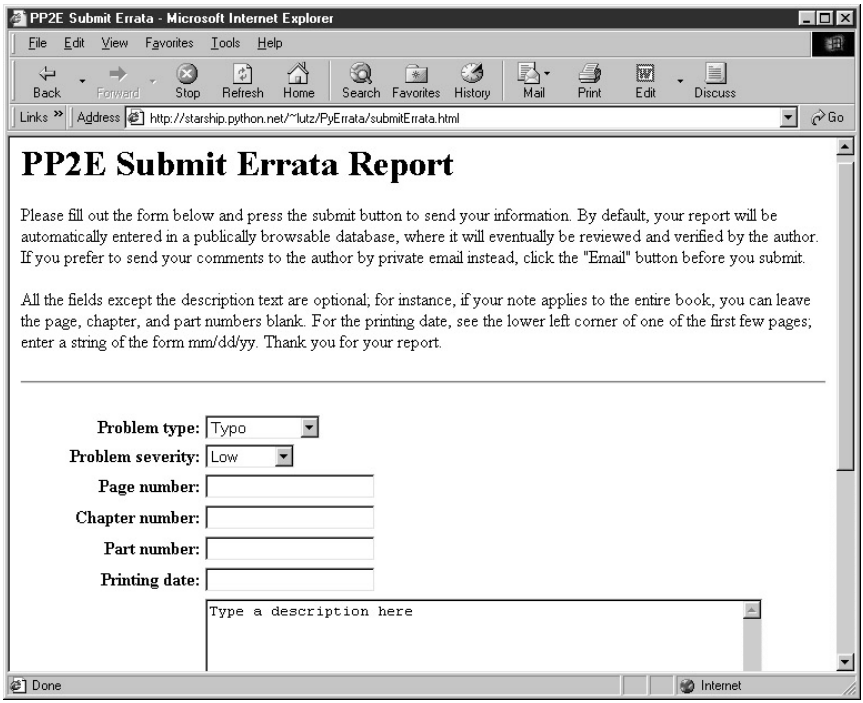


Рис. 14.20. Передача сообщения об ошибке, страница ввода (начало)

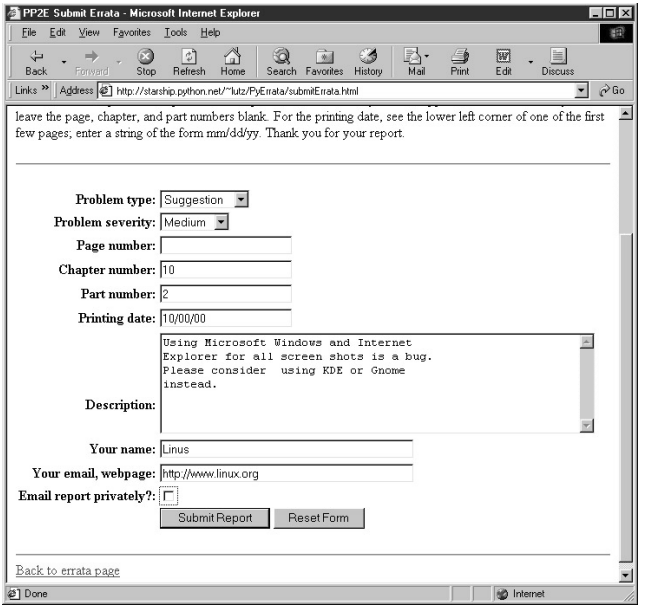


Рис. 14.21. Передача сообщения об ошибке, страница ввода (заполненная)

При нажатии кнопки передачи возвращается страница подтверждения, как и раньше (рис. 14.22), на этот раз с текстом, в котором выражается благодарность за сообщение об ошибке, а не комментарий.

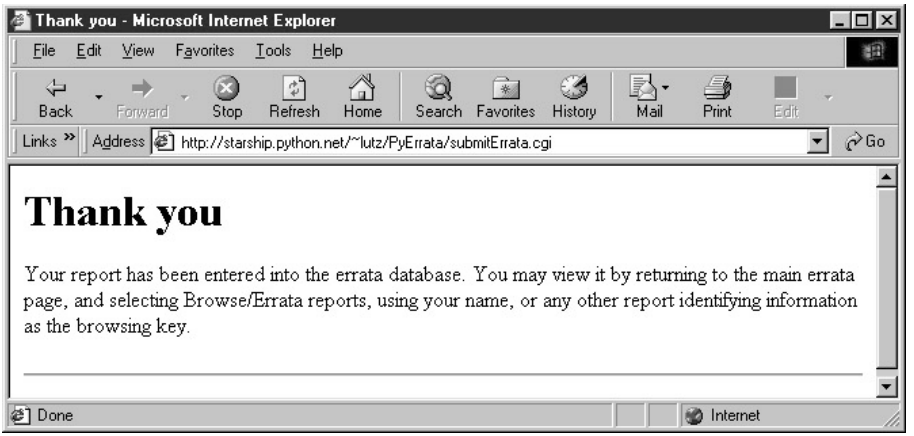


Рис. 14.22. Передача сообщения об ошибке, подтверждение

Как и прежде, можно проверить передачу сообщения с помощью страниц просмотра сразу после получения подтверждения. Вызовем страницу со списком по датам передачи и щелкнем по новой записи внизу (рис. 14.23). Наше сообщение извлекается из базы данных ошибок и выводится на новой странице (рис. 14.24). Обратите внимание, что поле «Page number» не отображается: оно было оставлено пустым в форме. При форматировании веб-страниц PyErrata выводит только непустые поля записей. Поскольку все записи обрабатываются общим способом, то же справедливо для сооб-

щений с комментариями; в основе своей PyErrata является очень общей системой, которой безразлично значение данных, хранящихся в записях.

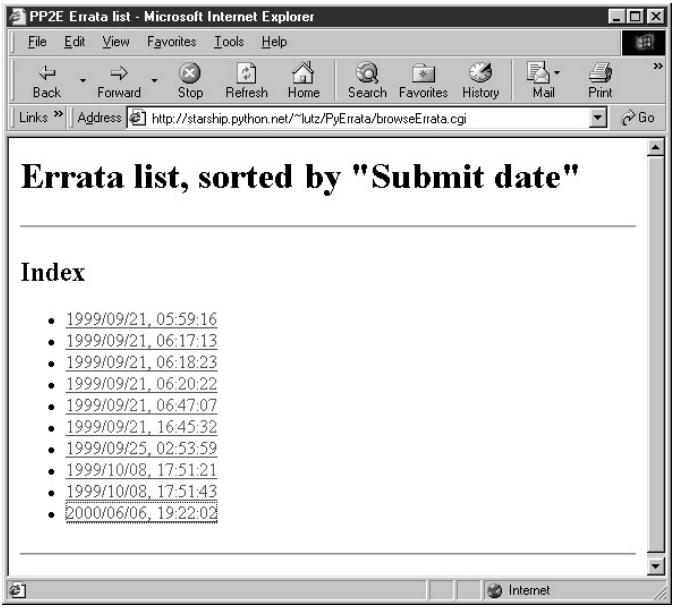


Рис. 14.23. Передача сообщения об ошибке, проверка результата (индекс)

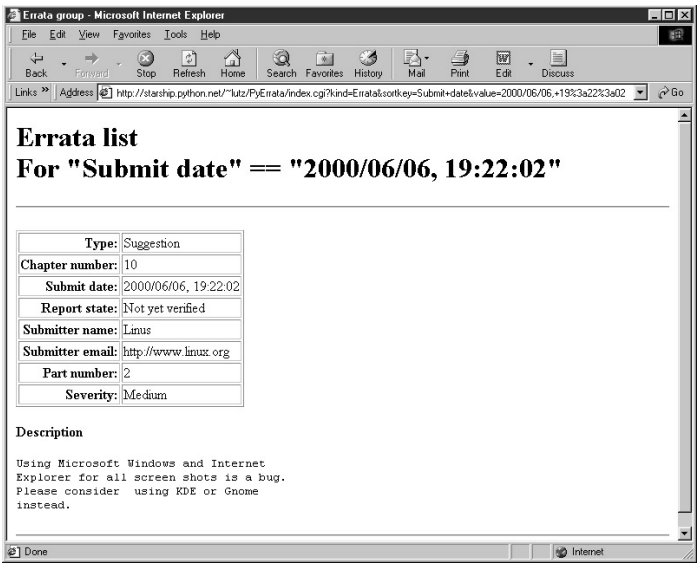


Рис. 14.24. Передача сообщения об ошибке, проверка результата (запись)

Поскольку не каждый захочет помещать данные в базу, которую с помощью броузера может посмотреть кто угодно, PyErrata позволяет также отправлять комментарии и сообщения об ошибках электронной почтой, а не помещать автоматически в базу дан-

ных. Если щелкнуть по флажку «Email report privately» внизу страницы перед передачей, детали сообщения будут отправлены мне электронной почтой (их поля окажутся в сообщении, направленном в мой почтовый ящик), и будет получена страница ответа, показанная на рис. 14.25.

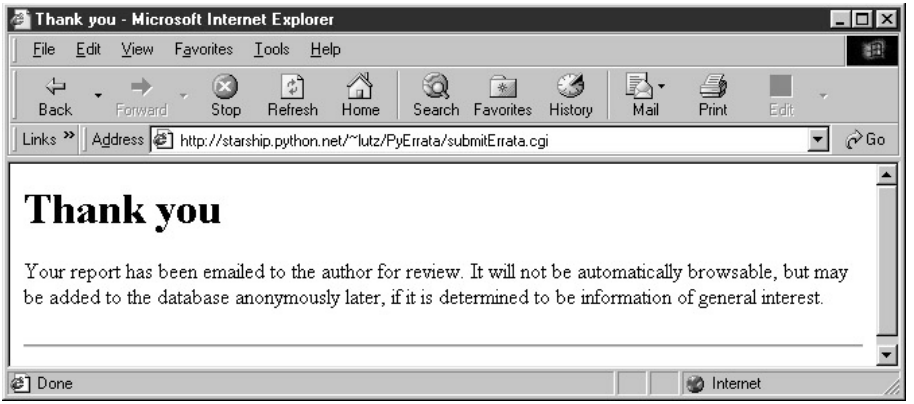


Рис. 14.25. Передача сообщения об ошибке, подтверждение для режима e-mail

Наконец, если каталог или файл shelve, представляющий базу данных, доступен по записи не для всех (вспомните, что сценарии CGI выполняются от имени «nobody»), наши сценарии не смогут сохранить новую запись. Python генерирует исключительную ситуацию, которая выводится в броузере клиента, потому что PyErrata направляет текст исключительной ситуации в `sys.stdout`. На рис. 14.26 показана страница с исключительной ситуацией, которую я получил, прежде чем сделал рассматриваемый каталог базы данных доступным по записи с помощью команды оболочки `chmod 777 DbaseFiles/errataDB`.

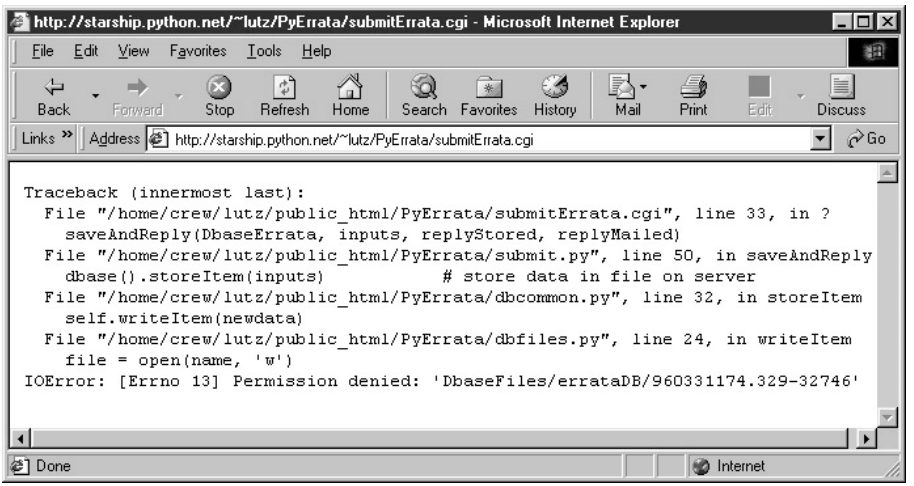


Рис. 14.26. Передача сообщения об ошибке, исключительная ситуация (требуется chmod 777 dir)

Реализация: передача сообщений с комментариями

Теперь, посмотрев на внешнее поведение PyErrata при операциях передачи, пора изучить их внутреннее устройство. Страницы верхнего уровня для передачи определяются статическими файлами HTML. В примере 14.8 показан файл страницы комментариев.

Пример 14.8. PP2E\Internet\Cgi-Web\PyErrata\submitComment.html

```
<html><body bgcolor="#FFFFFF">
<title>PP2E Submit Comment</title>
<h1>PP2E Submit Comment</h1>

<p>Please fill out the form below and press the submit button to
send your information.  By default, your report will be automatically
entered in a publically browsable database, where it will eventually
be reviewed by the author.  If you prefer to send your comments to the
author by private email instead, click the "Email" button before you
submit.  All the fields except the description text are optional.
Thank you for your report.
</p>

<hr>
<form method=POST action="submitComment.cgi">
  <table>
    <tr>
      <th align=right>Description:
      <td><textarea name="Description" cols=40 rows=10>Type your comment here
          </textarea>
    <tr>
      <th align=right>Your name:
      <td><input type=text size=35 name="Submitter name">
    <tr>
      <th align=right>Your email, webpage:
      <td><input type=text size=35 name="Submitter email">
    <tr>
      <th align=right>Email report privately?:
      <td><input type=checkbox name="Submit mode" value="email">
    <tr>
      <th></th>
      <td><input type=submit value="Submit Comment">
          <input type=reset  value="Reset Form">
  </table>
</form>

<hr>
<A href="pyerrata.html">Back to errata page</A>
</body></html>
```

Сценарий CGI, вызываемый при передаче формы из этого файла и приведенный в примере 14.9, осуществляет запись входных данных формы в базу данных и генерирует ответную страницу.

Пример 14.9. PP2E\Internet\Cgi-Web\PyErrata\submitComment.cgi

```
#!/usr/bin/python

DEBUG=0
if DEBUG:
    import sys
    sys.stderr = sys.stdout
    print "Content-type: text/html"; print
```

```
import traceback
try:
    from dbswitch import DbaseComment  # dbfiles или dbshelve
    from submit   import saveAndReply  # повторное использование логики сохранения
    replyStored = """
    Your comment has been entered into the comments database.
    You may view it by returning to the main errata page, and
    selecting Browse/General comments, using your name, or any
    other report identifying information as the browsing key."""

    replyMailed = """
    Your comment has been emailed to the author for review.
    It will not be automatically browsable, but may be added to
    the database anonymously later, if it is determined to be
    information of general use."""

    inputs = {'Description':'',     'Submit mode':'',
              'Submitter name':'',  'Submitter email':''}

    saveAndReply(DbaseComment, inputs, replyStored, replyMailed)

except:
    print "\n\n<PRE>"
    traceback.print_exc()
```

Не ищите здесь код для базы данных или генерации HTML: он весь выделен в модуль `submit`, показанный ниже, чтобы его можно было повторно использовать также для передачи сообщений об ошибках. Здесь мы просто передаем ему то, в чем разнятся передачи комментариев и ошибок: базы данных, ожидаемые поля ввода и текст ответа.

Как и раньше, из модуля switch берется объект интерфейса базы данных, чтобы выбрать используемый в настоящее время способ хранения. Индивидуальный текст для страниц подтверждения (`replyStored`, `replyMailed`) попадает в итоге на веб-страницы и может быть разным для каждой базы данных.

Словарь `inputs` в этом сценарии обеспечивает значения по умолчанию для отсутствующих полей и определяет формат записей комментариев в базе данных. На самом деле этот словарь хранится в базе данных: в модуле `submit` поля ввода из формы или явного URL сливаются с создаваемым здесь словарем `inputs`, а результат записывается в базу данных как запись.

Более точно, модуль `submit` проходит все ключи в `inputs` и выбирает значения этих ключей из объекта входных полей формы, если они там есть. В результате сценарий гарантирует наличие в записях базы данных комментариев всех полей, перечисленных в `inputs`, и никаких других. Поскольку все запросы передачи вызывают этот сценарий, это будет соблюдено, даже если передать в явном URL лишние поля: только поля из `inputs` будут сохранены в базе данных.

Обратите внимание, что почти весь этот сценарий заключен в оператор `try` с пустым предложением `except`. Этим гарантируется, что любая (не перехваченная) исключительная ситуация, которая может возникнуть во время выполнения нашего сценария, возвратится в этот `try` и выполнит его обработчик исключительной ситуации; в данном случае выполняется стандартный вызов `traceback.print_exc` для вывода деталей исключительной ситуации в веб-броузер в неформатированном (`<PRE>`) режиме.

Реализация: передача сообщений об ошибках

Страница верхнего уровня для передачи сообщений об ошибках, показанная на рис. 14.20 и 14.21, тоже выводится из файла статического HTML на сервере, приве-

денного в примере 14.10. Здесь больше полей ввода, но страница аналогична странице передачи комментариев.

Пример 14.10. PP2E\Internet\Cgi-Web\PyErrata\submitErrata.html

```
<html><body bgcolor="#FFFFFF">
<title>PP2E Submit Errata</title>
<h1>PP2E Submit Errata Report</h1>

<p>Please fill out the form below and press the submit button to
send your information.  By default, your report will be automatically
entered in a publically browsable database, where it will eventually
be reviewed and verified by the author.  If you prefer to send your
comments to the author by private email instead, click the "Email"
button before you submit.

<p>All the fields except the description text are optional;
for instance, if your note applies to the entire book, you can leave
the page, chapter, and part numbers blank.  For the printing date, see
the lower left corner of one of the first few pages; enter a string of
the form mm/dd/yy.  Thank you for your report.
</p>

<hr>
<form method=POST action="submitErrata.cgi">
  <table>
    <tr>
      <th align=right>Problem type:
      <td><select name="Type">
          <option>Typo
          <option>Grammar
          <option>Program bug
          <option>Suggestion
          <option>Other
      </select>
    <tr>
      <th align=right>Problem severity:
      <td><select name="Severity">
          <option>Low
          <option>Medium
          <option>High
          <option>Unknown
      </select>
    <tr>
      <th align=right>Page number:
      <td><input type=text name="Page number">
    <tr>
      <th align=right>Chapter number:
      <td><input type=text name="Chapter number">
    <tr>
      <th align=right>Part number:
      <td><input type=text name="Part number">
    <tr>
      <th align=right>Printing date:
      <td><input type=text name="Printing date">
    <tr>
      <th align=right>Description:
      <td><textarea name="Description" cols=60 rows=10>Type a description here
          </textarea>
    <tr>
```

```
      <th align=right>Your name:
      <td><input type=text size=40 name="Submitter name">
    <tr>
      <th align=right>Your email, webpage:
      <td><input type=text size=40 name="Submitter email">
    <tr>
      <th align=right>Email report privately?:
      <td><input type=checkbox name="Submit mode" value="email">
    <tr>
      <th></th>
      <td><input type=submit value="Submit Report">
          <input type=reset  value="Reset Form">
  </table>
</form>

<hr>
<A href="pyerrata.html">Back to errata page</A>
</body></html>
```

Сценарий, запускаемый формой на этой странице и показанный в примере 14.11, тоже весьма похож на сценарий `submitComment` **из примера 14.9. Поскольку в обоих сценариях просто используется логика, выделенная в модуль** `submit`**, все, что нам здесь требуется, это передать соответствующим образом подготовленный текст страниц подтверждения и ожидаемые поля ввода. Как и прежде, действительные входные данные CGI сливаются со словарем** `inputs` **в сценарии, образуя запись базы данных; сохраняемая запись будет содержать точно те поля, которые здесь перечислены.**

Пример 14.11. PP2E\Internet\Cgi-Web\PyErrata\submitErrata.cgi

```
#!/usr/bin/python

DEBUG=0
if DEBUG:
    import sys
    sys.stderr = sys.stdout
    print "Content-type: text/html"; print

import traceback
try:
    from dbswitch import DbaseErrata   # dbfiles или dbshelve
    from submit   import saveAndReply  # повторное использование логики сохранения
    replyStored = """
    Your report has been entered into the errata database.
    You may view it by returning to the main errata page, and
    selecting Browse/Errata reports, using your name, or any
    other report identifying information as the browsing key."""

    replyMailed = """
    Your report has been emailed to the author for review.
    It will not be automatically browsable, but may be added to
    the database anonymously later, if it is determined to be
    information of general interest."""

    # При записи добавляются 'Report state' и 'Submit date'

    inputs = {'Type':'',             'Severity':'',
              'Page number':'',      'Chapter number':'',    'Part number':'',
              'Printing Date':'',    'Description':'',       'Submit mode':'',
              'Submitter name':'',   'Submitter email':''}

    saveAndReply(DbaseErrata, inputs, replyStored, replyMailed)
```

```
except:
    print "\n\n<pre>"
    traceback.print_exc()
```

Общий вспомогательный модуль submit

Оба вида сообщений – с комментариями и с ошибками – в конечном итоге вызывают функции модуля из примера 14.12, чтобы сделать запись в базе данных и сгенерировать страницу ответа. Главная задача этого модуля – слить реальные данные ввода CGI со словарем ожидаемых входных данных и передать результат в базу данных или электронную почту. Основные идеи кода этого модуля мы уже описывали, поэтому добавить здесь можно немного.

Обратите, однако, внимание, что передача в режиме электронной почты (включаемом при установке флажка e-mail на странице передачи) использует вызов команды оболочки `os.popen`; письма, поступающие в мой почтовый ящик, содержат по одной строчке для каждого непустого поля сообщения. На моем веб-сервере, работающем под Linux, это действует, но лучше переносимыми являются другие почтовые схемы, например с использованием модуля `smptlib` (описанного в главе 11 «Сценарии на стороне клиента»).

Пример 14.12. PP2E\Internet\Cgi-Web\PyErrata\submit.py

```
#########################################################################################
# При запросе передачи: записать данные или отправить их по почте, передать ответную страницу;
# данные сообщений записываются в словарях в базе данных; обязательным полем является
# description (если оно пусто, возвращается страница сообщения об ошибке), хотя механизм
# базы данных может обрабатывать пустые поля description – нет смысла сообщать об ошибке
# и не описывать ее
#########################################################################################

import cgi, os, sys, string
mailto = 'lutz@rmi.net'              # или lutz@starship.python.net
sys.stderr = sys.stdout              # вывод ошибок в броузер
print "Content-type: text/html\n"

thankyouHtml = """
<TITLE>Thank you</TITLE>
<H1>Thank you</H1>
<P>%s</P>
<HR>"""

errorHtml = """
<TITLE>Empty field</TITLE>
<H1>Error: Empty %s</H1>
<P>Sorry, you forgot to provide a '%s' value.
Please go back to the prior page and try again.</P>
<HR>"""

def sendMail(inputs):                              # послать e-mail автору
    text = ''                                      # или действие формы 'mailto:'
    for key, val in inputs.items():                # или smtplib.py, или sendmail
        if val != '':
            text = text + ('%s = %s\n' % (key, val))
    mailcmd = 'mail -s "PP2E Errata" %s' % mailto
    os.popen(mailcmd, 'w').write(text)

def saveAndReply(dbase, inputs, replyStored, replyMailed):
    form = cgi.FieldStorage()
    for key in form.keys():
```

```
        if key in inputs.keys():
            inputs[key] = form[key].value       # выбрать введенные поля

    required = ['Description']
    for field in required:
        if string.strip(inputs[field]) == '':
            print errorHtml % (field, field)    # послать в броузер страницу ошибки
            break
    else:
        if inputs['Submit mode'] == 'email':
            sendMail(inputs)     # послать данные автору по e-mail
            print thankyouHtml % replyMailed
        else:
            dbase().storeItem(inputs)           # записать данные в файл на сервере
            print thankyouHtml % replyStored
```

Этот модуль использует для сохранения словарей записей еще один интерфейс базы данных: `dbase().storeItem(inputs)`. Однако чтобы разобраться, какая обработка происходит при этом вызове, нужно перейти к следующему разделу.

Еще одно предупреждение об избыточности: список ожидаемых полей в словарях `inputs` сценариев передачи тот же самый, что в списке входных полей файлов HTML для передачи. В принципе, мы снова могли бы генерировать список полей файла HTML по данным в общем модуле и устранить избыточность. Однако здесь этот прием применить не столь просто, потому что в файле HTML требуется описание для каждого поля.

Интерфейсы баз данных PyErrata

Теперь, познакомившись с интерфейсами пользователя и реализациями верхнего уровня операций просмотра и передачи, опустимся в этом разделе на один уровень абстракции и перейдем к третьей и последней из главных функциональных областей системы PyErrata.

По сравнению с другими системами в этой части книги, одной из наиболее отличительных технических характеристик PyErrata является необходимость управлять постоянно хранящимися данными. Информацию, переданную пользователями, нужно занести в базу данных, чтобы в дальнейшем просматривать ее. PyErrata хранит сообщения в виде словарей и содержит логику, поддерживающую два способа хранения данных в базе – плоские файлы сериализации (pickle files) и полки (shelves), а также средства синхронизации доступа к данным.

Призрак одновременных обновлений

Есть несколько способов, с помощью которых сценарии Python могут хранить данные постоянно: файлы, сериализация объектов, полки объектов (файлы shelve), настоящие базы данных и т. д. На самом деле глава 16 посвящена исключительно этой теме и обеспечивает более глубокое освещение, чем нам здесь требуется.[1] Все эти способы хранения действуют также в контексте сценариев CGI на стороне сервера, но в окру-

[1] Но посмотрите главу 16, чтобы получить некоторую подготовку по этой теме. Данная глава представляет и использует только простейшие интерфейсы модулей `pickle` и `shelve`, а обсуждение большинства деталей интерфейсов модулей откладывается до следующей главы.

жении CGI почти автоматически возникает новая проблема: *одновременных обновлений*. Поскольку модели CGI внутренне присуща параллельность, сценарии должны обеспечить должную синхронизацию записи и чтения в базе данных, чтобы избежать нарушения целостности данных и незавершенных записей.

И вот почему. PyErrata позволяет читателю посетить сайт и передать сообщение или прочесть предыдущие сообщения. Но в контексте веб-приложения невозможно узнать, сколько читателей одновременно осуществляет передачу или просмотр сообщений: любое число людей может одновременно нажать кнопку submit формы. Мы видели, что передача формы обычно заставляет сервер HTTP породить новый процесс для обработки запроса. Поскольку все эти обрабатывающие процессы выполняются параллельно, то если сто пользователей одновременно нажмут submit, на сервере будут параллельно работать сто процессов сценариев CGI, и все они могут попытаться одновременно обновить (или прочесть) базу данных сообщений.

Из-за возможности такой параллельной передачи информации программы сервера, работающие с базой данных, должны каким-то образом обеспечить поочередное обновление базы данных, иначе база данных может быть повреждена. С ростом числа пользователей сайта увеличивается вероятность возникновения двух конкретных ситуаций:

- *Одновременная запись*: Если два процесса пытаются одновременно обновить один и тот же файл, это может окончиться тем, что часть новых данных одного процесса перемешается с данными другого процесса, часть данных одного процесса будет потеряна или записываемые данные будут искажены еще каким-то образом.
- *Одновременные чтение и запись*: Аналогично, если некоторый процесс попытается прочесть запись, которую в это время сохраняет другой процесс, он может получить незавершенное сообщение. В сущности, базой данных нужно управлять как ресурсом, совместно используемым всеми возможными процессами обработчиков CGI, независимо от того, производят ли они обновление.

Ограничения меняются в зависимости от носителя базы данных, и в то время как обычно допускается одновременное *чтение* базы данных несколькими процессами, процессы *записи* (и обновления в целом) почти всегда должны получать исключительный доступ к совместно используемой базе данных. Есть ряд способов обеспечить безопасный доступ к базе данных в среде, где возможно одновременное обращение к ней, такой как сайты, использующие CGI:

Системы баз данных

Большинство развитых систем баз данных (таких, как Sybase, Oracle, mySql) обеспечивает поддержку одновременного доступа, но работа с ними усложняет приложение.

Централизованные серверы баз данных

Можно также координировать доступ к совместно используемым хранилищам данных, направляя все запросы данных постоянно выполняющейся управляющей программе, которую вы реализуете сами. Имеется в виду, что когда сценарию CGI требуется обратиться к базе данных, он должен запросить доступ у программы сервера данных через коммуникационный протокол, такой как вызовы сокетов.

Соглашения по именам файлов

Если можно хранить каждую запись базы данных в отдельном плоском файле, то иногда удается избежать проблем одновременного доступа или свести их к минимуму путем присвоения каждому плоскому файлу особого имени. Например, если имя файла каждой записи содержит время создания файла и ID создавшего его процесса, оно окажется уникальным для всех практических задач, потому что данный процесс обновляет только один конкретный файл. Такая система опирает-

ся на файловую систему операционной системы, чтобы сделать записи отличными друг от друга путем сохранения их в файлах с уникальными именами.

Протоколы блокировки файлов

Если вся база данных физически хранится в одном файле, можно с помощью средств операционной системы блокировать файл на время обновления. На серверах Unix и Linux монопольная блокировка файла приостанавливает другие процессы, которым он требуется, пока блокировка не будет снята; при согласованном использовании всеми процессами такой механизм автоматически синхронизирует обращения к базе данных. Объекты shelve в Python поддерживают одновременное чтение, но не одновременное обновление, поэтому нужно самостоятельно добавлять блокировки, чтобы использовать файлы shelve в качестве динамических хранилищ данных сценариями CGI.

В этом разделе мы реализуем для PyErrata две последние схемы, чтобы проиллюстрировать основы одновременного доступа к данным.

Структура хранилища базы данных

Прежде всего, разберемся с тем, что же в действительности хранит система. Если перелистать книгу назад к рис. 14.1, то можно увидеть два каталога баз данных верхнего уровня: *DbaseShelve* (для механизма shelve) и *DbaseFiles* (для файлового хранения). В каждом из этих каталогов особое содержимое.

База данных shelve

Для баз данных, основанных на shelve, предназначен каталог *DbaseShelve*, содержимое которого показано на рис. 14.27. Файлы `commentDB` и `errataDB` служат объектами shelve для хранения сообщений, а файлы *.lck* и *.log* служат файлами блокировки и журнала, генерируемыми системой. Для установки системы с нуля первоначально требуются только два файла *.lck* (которые могут быть пустыми); система создаст файлы shelve и журнала, когда будут сохранены записи.

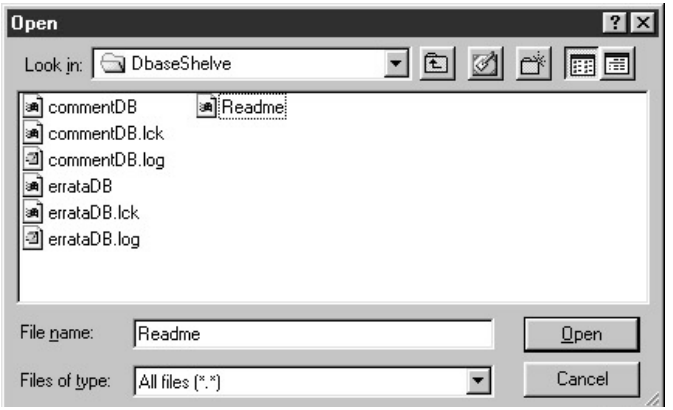


Рис. 14.27. Содержимое каталога для использующей полки базы данных PyErrata

Модуль Python `shelve` более подробно изучается в следующей части книги, а те части его, которые используются в этой главе, просты. Вот основные интерфейсы `shelve`, используемые в этом примере:

```
import shelve                  # загрузить стандартный модуль shelve
```

```
dbase = shelve.open('filename') # открыть (создать) файл shelve
dbase['key'] = object           # запись почти любого объекта в файл shelve
object = dbase['key']           # в дальнейшем получить объект из shelve
dbase.keys()                    # список ключей, хранящихся в shelve
dbase.close()                   # закрыть файл shelve
```

Иными словами, объекты shelve напоминают словари объектов Python, отображенные во внешний файл, и потому сохраняются в промежутке между запусками программы. Объекты в shelve запоминаются и позднее загружаются по ключу. В действительности не будет ошибкой представлять себе объекты shelve как словари, сохраняющиеся после выхода из программы и требующие явного открытия.

Подобно словарям, у каждого отдельного значения, хранящегося в shelve, должен быть *уникальный ключ*. Так как в комментарии или сообщении об ошибке нет поля, которое с уверенностью можно было бы считать неповторяющимся для всех сообщений, приходится генерировать такое поле самостоятельно. Время передачи почти уникально, но нет гарантии, что два пользователя (а следовательно, два процесса) не передадут сообщение в одну и ту же секунду.

Чтобы назначить каждой записи уникальное место в shelve, система генерирует для каждой из них уникальную строку ключа, которая содержит время передачи (секунды с начала «эпохи» Unix в виде значения с плавающей точкой) и ID процесса сценария CGI, производящего запись. Поскольку значения словаря, хранящиеся в shelve, содержат всю интересующую нас информацию сообщения, от ключей требуется лишь уникальность, а не присутствие какого-то смысла. Записи загружаются путем бездумного перебора по списку ключей shelve.

Помимо генерации уникальных ключей записей объекты shelve должны обеспечивать *одновременные обновления*. Так как shelve отображаются в один файл файловой системы (в данном случае `errataDB` или `commentDB`), необходимо синхронизировать доступ к ним в среде, где возможны параллельные процессы, например в сценариях CGI.

В своем настоящем виде модуль Python `shelve` поддерживает одновременное чтение, но не одновременное обновление, поэтому такую возможность нужно добавить самостоятельно. Реализация полочной схемы хранения базы данных в PyErrata использует блокировки файлов *.lck*, чтобы обеспечить программам записи (процессам передачи) получение исключительного доступа к shelve перед выполнением обновления. Параллельно может выполняться любое количество процессов чтения, но процессы записи должны выполняться по одному и блокировать все другие процессы – чтения и записи – во время обновления shelve.

Обратите внимание на использование при блокировке отдельных файлов *.lck* вместо блокирования самого файла shelve. В одних системах объекты shelve отображаются в несколько файлов, а в других (например, GDBM), блокировки файла shelve зарезервированы для использования самой файловой системой DBM. Использование своего файла блокировки справляется с такими ограничениями и лучше переносимо между разными реализациями DBM.

База данных на плоских файлах

Иначе обстоит дело для баз данных на плоских файлах. На рис. 14.28 показано содержимое подкаталога базы данных ошибок, основанной на файлах, *DbaseFiles/errataDB*. В этой схеме каждое сообщение хранится в отдельном плоском файле с уникальным именем, содержащем сериализованный словарь данных сообщения. Аналогичный каталог, *DbaseFiles/commentDB*, существует для комментариев. Чтобы начать работу с нуля, требуется лишь существование двух подкаталогов; файлы добавляются по мере передачи сообщений.

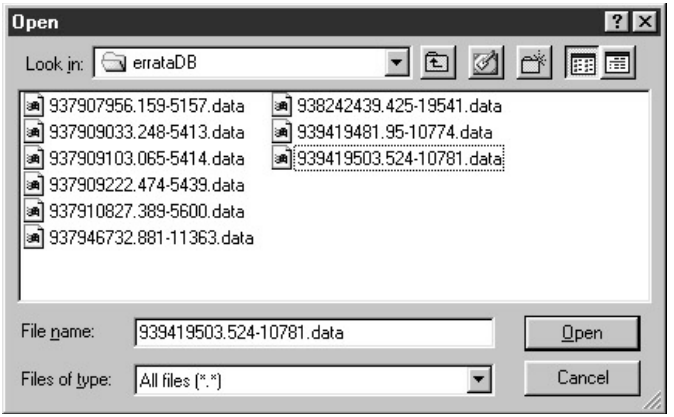


Рис. 14.28. Содержимое каталога базы данных PyErrata, основанной на файлах

Специальный объект Python pickler преобразует («сериализует») объекты, находящиеся в памяти, в особым образом кодируемые строки и восстанавливает их обратно за один шаг и потому оказывается удобным для сохранения сложных объектов типа словарей, которые используются PyErrata для представления записей сообщений.[1] Мы будем также углубленно изучать модуль `pickle` в части IV «Разное», но те его интерфейсы, которые используются PyErrata, тоже просты:

```
pickle.dump(object, outputfile)          # сохранить объект в файле
object = pickle.load(inputfile)          # загрузить объект обратно из файла
```

Для плоских файлов генерируемый системой и присваиваемый записи *ключ* следует тому же формату, что и для shelve, но теперь он используется для образования имени файла с сообщением. Благодаря этому ключи записей более очевидны (мы видим их при просмотре каталога), но по-прежнему не передают какой-либо значимой информации. От них требуется только уникальность для каждой сохраняемой записи, чтобы получить уникальный файл. В этой схеме хранения записи обрабатываются путем обхода списка содержимого каталога, возвращаемого стандартным вызовом `glob.glob` для шаблона имени `*.data` (см. главу 2 «Системные инструменты», чтобы вспомнить модуль `glob`).

В некотором смысле этот подход на основе плоского файла использует в качестве полки файловую систему и полагается на операционную систему при разделении записей в виде файлов. При этом также не приходится особенно беспокоиться о проблемах *одновременного доступа*; поскольку создаваемые имена файлов обеспечивают хранение каждого сообщения в собственном отдельном файле, два разных процесса передачи не

[1] В PyErrata можно было бы также просто записывать словари записей сообщений в файлы по одному ключу и его значению в строчке, а потом расщеплять строки для восстановления записей. Можно было бы и просто преобразовывать словарь записи в строковое представление с помощью встроенной функции `str`, вручную записывать эту строку в файл и впоследствии преобразовывать строку обратно в словарь с помощью встроенной функции `eval` (что может оказаться более медленным из-за общих накладных расходов на синтаксический анализ в `eval`). Однако, как будет показано в следующей части книги, сериализация объектов является значительно более мощным и общим подходом к хранению объектов – она справляется также с такими вещами, как объекты экземпляров классов и совместно используемые или рекурсивные ссылки на объекты. Смотрите аналогичные темы для классов оболочек таблиц в примере PyForm главы 16.

могут писать в один и тот же файл одновременно. Более того, можно читать одно сообщение, в то время как создается другое: это действительно разные файлы.

Однако по-прежнему нужно следить за тем, чтобы файл не был виден для чтения в содержимом каталога, пока его запись не завершена, иначе можно прочесть файл, записанный не полностью. На практике такой случай маловероятен – он может произойти только тогда, когда пишущий процесс не завершил свою работу к тому времени, когда читающий процесс доберется до этого файла в своем списке содержимого каталога. Во избежание проблем сценарии передачи сначала записывают данные во временный файл и только после завершения записи дают временному файлу действительное имя `*.data`.

Переключатель баз данных

Перейдем к листингам кода. Первый модуль базы данных в примере 14.13 просто осуществляет выбор между механизмом, основанным на файлах, и механизмом, основанным на shelve. Выбор осуществляется только здесь, чтобы он не влиял на другие файлы при изменении схемы хранения.

Пример 14.13. PP2E\Internet\Cgi-Web\PyErrata\dbswitch.py

```
##############################################################################
# Для проверки альтернативных способов применения баз данных; так как сценарии cgi
# для просмотра, передачи и составления индекса импортируют имена баз данных
# только отсюда, они получат то, что загрузит этот модуль; иными словами,
# для переключения способа просто поменяйте здесь импорт; в конечном счете,
# можно было бы вообще убрать этот интерфейсный модуль и непосредственно загрузить
# модуль самого лучшего способа, но самый лучший определяется моделью применения;
##############################################################################

#
# один каталог для каждой базы, один плоский файл сериализации для каждой передачи

from dbfiles import DbaseErrata, DbaseComment

#
# одна полка для каждой базы, один ключ для сообщения с мьютексными блокировками обновления

# from dbshelve import DbaseErrata, DbaseComment
```

Классы для файлов и объектов shelve, специфичные для способа хранения

Следующие два модуля реализуют объекты доступа к базе данных, основанные на файлах и на shelve. Определяемые ими классы суть объекты, получаемые и используемые в сценариях просмотра и передачи. Оба они в действительности являются подклассами более общего класса из `dbcommon`; в примере 14.14 мы реализуем методы, определяющие поведение, специфичное для схемы хранения, но большую часть работы выполняет суперкласс.

Пример 14.14. PP2E\Internet\Cgi-Web\PyErrata\dbfiles.py

```
##############################################################################################
# Записать каждый элемент в отдельный плоский файл, сериализованным; в dbcommon предполагается,
# что записи - словари, здесь - нет; chmod 666 разрешит доступ администратору (иначе - владелец
# 'nobody'); нюанс: уникальные имена файлов не дают одновременно писать в один файл, но все же
# может быть попытка чтения (броузером) файла, когда он записан еще не до конца, если glob.glob
# вернет имя созданного, но незавершенного файла; это маловероятно (файл должен оказаться
```

```
# незавершенным по истечении времени перехода от glob к unpickle), но чтобы избежать
# этой опасности, файлы создаются с временными именами и получают свое настоящее имя только
# после завершения записи и закрытия; сценарии cgi с постоянными данными подвержены проблеме
# параллельного обновления, потому что несколько сценариев cgi могут выполняться одновременно;
###############################################################################################

import dbcommon, pickle, glob, os

class Dbase(dbcommon.Dbase):
    def writeItem(self, newdata):
        name = self.dirname + self.makeKey()
        file = open(name, 'w')
        pickle.dump(newdata, file)          # записать в новый файл
        file.close()
        os.rename(name, name+'.data')       # видим для glob
        os.chmod(name+'.data', 0666)        # владелец 'nobody'

    def readTable(self):
        reports = []
        for filename in glob.glob(self.dirname + '*.data'):
            reports.append(pickle.load(open(filename, 'r')))
        return reports

class DbaseErrata(Dbase):
    dirname = 'DbaseFiles/errataDB/'

class DbaseComment(Dbase):
    dirname = 'DbaseFiles/commentDB/'
```

Модуль интерфейса к shelve, приведенный в примере 14.15, предоставляет интерфейс с теми же методами, но их реализация ориентирована на shelve. В его класс также встраивается (mixes in) класс взаимного исключения для обеспечения блокировки файлов; код этого класса мы изучим несколькими страницами ниже.

Обратите внимание на то, что этот модуль расширяет `sys.path` таким образом, что специфический для платформы модуль `FCNTL` (описываемый далее в этой главе) становится видимым средствам блокировки файлов. Это требуется только в контексте сценария CGI, потому что путь поиска модулей, который дается пользователю CGI «nobody» не содержит каталога специфических для платформы модулей расширения. Классы файлов и shelve устанавливают для вновь создаваемых файлов права с восьмеричной маской 0666, благодаря чему другие пользователи, помимо «nobody», могут выполнять чтение и запись. Если вы забыли, кто такой «nobody», посмотрите обсуждение проблем прав доступа и владения в этой и двух предыдущих главах.

Пример 14.15. PP2E\Internet\Cgi-Web\PyErrata\dbshelve.py

```
###############################################################################################
# Сохранение элементов на полке с блокировкой файлов при записи; в dbcommon предполагается,
# что записи - словари, здесь - нет; вызов chmod предполагает один файл для каждого объекта
# shelve (например, gdbm); shelve разрешает одновременное чтение, но если какая-то программа
# выполняет запись, чтение и запись запрещены другим программам, в результате получается
# блокировка всех операций загрузки/записи; необходимо chmod 0666, иначе записывать
# сможет только 'nobody'; этот файл не знает о fcntl, а mutex не знает о сценариях cgi -
# один из двух должен добавить путь к модулю FCNTL для использования только сценариями
# cgi (здесь); мы обходим механизм блокировки, который может быть у используемой системы БД,
# т.к. блокируем собственный файл, а не файл БД перед проведением операций с БД;
# допускается параллельное чтение, но для записи требуется исключительный доступ к shelve;
# блокирующие вызовы в блоке MutexCntl, затем при необходимости вызывавшие возобновляются;
###############################################################################################
```

```
# cgi выполняется как 'nobody' без
# следующих путей по умолчанию
import sys
sys.path.append('/usr/local/lib/python1.5/plat-linux2')

import dbcommon, shelve, os
from Mutex.mutexcntl import MutexCntl

class Dbase(MutexCntl, dbcommon.Dbase):           # встроить mutex, dbcommon
    def safe_writeItem(self, newdata):
        dbase = shelve.open(self.filename)        # получить исключительный доступ: обновление
        dbase[self.makeKey()] = newdata           # безопасное сохранение на полке
        dbase.close()
        os.chmod(self.filename, 0666)             # чтобы другие не могли изменить

    def safe_readTable(self):
        reports = []                              # получить совместный доступ: загрузка
        dbase = shelve.open(self.filename)        # записи не будет
        for key in dbase.keys():
            reports.append(dbase[key])            # загрузка данных, надежная
        dbase.close()
        return reports

    def writeItem(self, newdata):
        self.exclusiveAction(self.safe_writeItem, newdata)

    def readTable(self):
        return self.sharedAction(self.safe_readTable)

class DbaseErrata(Dbase):
    filename = 'DbaseShelve/errataDB'

class DbaseComment(Dbase):
    filename = 'DbaseShelve/commentDB'
```

Класс интерфейса базы данных верхнего уровня

Здесь мы добрались до интерфейсов баз данных верхнего уровня, к которым фактически обращаются наши сценарии CGI. Класс примера 14.16 является «абстрактным» в том смысле, что самостоятельно он ничего не может сделать. Необходимо предоставить и создать экземпляры подклассов, в которых определяются специфические для способа хранения методы, а не создавать непосредственно экземпляры этого класса.

На самом деле в этом классе используемая схема хранения данных умышленно оставляется неопределенной, и если подкласс не восполняет необходимые детали, возбуждаются ошибки утверждения (assertion errors). В эту модель интерфейса класса верхнего уровня можно включить любой класс для конкретного способа хранения, который предоставляет методы `writeItem` и `readTable`. В это число входят классы, работающие с плоскими файлами, полками, и другие спецификации, которые могут быть добавлены в будущем (например, схемы, взаимодействующие с полноценными объектными или SQL-базами данных либо сохраняющие данные в постоянных серверах).

В некотором смысле подклассы играют здесь роль встроенных объектов компонентов: они просто должны предоставить ожидаемые интерфейсы. Так как интерфейс верхнего уровня был выделен в этот одиночный класс, поменять используемую схему хранения можно путем выбора другого подкласса с иной схемой хранения (как в `dbswitch`); обращения к базе данных верхнего уровня останутся неизменными. Более того, изменения и оптимизация интерфейсов верхнего уровня может коснуться только этого файла.

Так как этот суперкласс является общим для классов, специфичных для схем хранения, в нем также определены методы генерации ключей записей и в новые записи вставляются общие генерируемые атрибуты (дата передачи, начальное состояние сообщения) перед тем, как их сохранить.

Пример 14.16. PP2E\Internet\Cgi-Web\PyErrata\dbcommon.py

```
################################################################################
# Абстрактный суперкласс с логикой совместного доступа к базе данных;
# сохраняемые записи предположительно имеют вид словарей (или другого отображения),
# по одному ключу в поле; используемая база данных не определена;
# суперклассы: определяют надлежащие writeItem и readTable для используемого
# файлового носителя - плоские файлы, shelve и т.д.
# нюанс: добавляемое здесь поле 'Submit date' можно хранить как кортеж, и будет действовать
# вся логика сортировки/выбора; но т.к. эти значения могут быть встроены в строку url,
# мы не хотим преобразовывать строку в кортеж с помощью eval в index.cgi;
# для согласованности и безопасности мы преобразовываем в строки здесь;
# если бы не проблема с url, кортежи можно использовать как ключи словарей;
# для сортировки в строке времени должны использоваться колонки фиксированной ширины;
# в будущих версиях этот интерфейс, возможно, будет оптимизирован;
################################################################################

import time, os

class Dbase:

    # запись

    def makeKey(self):
        return "%s-%s" % (time.time(), os.getpid())

    def writeItem(self, newdata):
        assert 0, 'writeItem must be customized'

    def storeItem(self, newdata):
        secsSinceEpoch          = time.time()
        timeTuple               = time.localtime(secsSinceEpoch)
        y_m_d_h_m_s             = timeTuple[:6]
        newdata['Submit date']  = '%s/%02d/%02d, %02d:%02d:%02d' % y_m_d_h_m_s
        newdata['Report state'] = 'Not yet verified'
        self.writeItem(newdata)

    # загрузка

    def readTable(self):
        assert 0, 'readTable must be customized'

    def loadSortedTable(self, field=None):    # возвращает простой список
        reports = self.readTable()            # упорядоченный по значению field
        if field:
            reports.sort(lambda x, y, f=field: cmp(x[f], y[f]))
        return reports

    def loadIndexedTable(self, field):
        reports = self.readTable()
        index = {}
        for report in reports:
          try:
            index[report[field]].append(report) # группировка по значениям поля
          except KeyError:
            index[report[field]] = [report]     # добавить сначала по этому ключу
```

```
    keys = index.keys()
    keys.sort()                              # отсортированные ключи, словарь групп
    return keys, index
```

Взаимные исключения для файлов shelve

Наконец мы добрались до самого низа иерархии кода PyErrata: кода, в котором инкапсулированы блокировки файлов для синхронизации доступа к полкам. Класс, приведенный в примере 14.17, предоставляет средства синхронизации операций с помощью блокировки файла, имя которого предоставляется системами, использующими этот класс.

Он содержит методы для блокировки и разблокировки файла, а также экспортирует методы более высокого уровня для выполнения вызовов функций в монопольном или совместном режиме. Метод `sharedAction` **используется для выполнения операций чтения, а** `exclusiveAction` **обрабатывает операции записи. Параллельно может выполняться любое число совместных (*shared*) действий, но монопольные (*exclusive*) действия происходят самостоятельно и блокируют все остальные запросы действий в параллельных процессах. Оба типа действий выполняются внутри оператора** `try-finally`**, что гарантирует снятие с файлов блокировок при выходе из действий, нормальным или иным способом.**

Пример 14.17. PP2E\Internet\Cgi-Web\PyErrata\Mutex\mutexcntl.py

```
##############################################################################################
# Полезный в целом встраиваемый класс, выделенный поэтому в отдельный модуль;
# требует установки атрибута self.filename и предполагает существование файла
# self.filename+'.lck'; установка mutexcntl.debugMutexCntl переключает
# запись в журнал; сообщения о блокировках записываются в self.filename+'.log';
##############################################################################################

import fcntl, os, time
from FCNTL import LOCK_SH, LOCK_EX, LOCK_UN

debugMutexCntl = 1
processType = {LOCK_SH: 'reader', LOCK_EX: 'writer'}

class MutexCntl:
    def lockFile(self, mode):
        self.logPrelock(mode)
        self.lock = open(self.filename + '.lck')    # блокировать файл в этом процессе
        fcntl.flock(self.lock.fileno(), mode)       # ждет блокировки при необходимости
        self.logPostlock()

    def lockFileRead(self):                  # разрешить > 1 чтения: совместные
        self.lockFile(LOCK_SH)               # ждать при наличии блокировки записи

    def lockFileWrite(self):                 # пишущий процесс получает монопольную блокировку
        self.lockFile(LOCK_EX)               # ждать при любой блокировке: r или w

    def unlockFile(self):
        self.logUnlock()
        fcntl.flock(self.lock.fileno(), LOCK_UN)    # разблокировать для других процессов

    def sharedAction(self, action, *args):   # интерфейс более высокого уровня
        self.lockFileRead()                  # остановка, если блокировка по записи
        try:
            result = apply(action, args)     # любое число совместных одновременно
        finally:                             # но никаких монопольных действий
            self.unlockFile()                # разрешить выполнение новых процессов записи
```

```
        return result

    def exclusiveAction(self, action, *args):
        self.lockFileWrite()                # приостановка, если есть другие блокировки
        try:
            result = apply(action, args)    # нет перекрытия с другими действиями
        finally:
            self.unlockFile()               # разрешить новые процессы чтения/записи
        return result

    def logmsg(self, text):
        if not debugMutexCntl: return
        log = open(self.filename + '.log', 'a')       # добавить в конец
        log.write('%s\t%s\n' % (time.time(), text))   # вывод не перезаписывает
        log.close()                                   # но может перемешиваться

    def logPrelock(self, mode):
        self.logmsg('Requested: %s, %s' % (os.getpid(), processType[mode]))
    def logPostlock(self):
        self.logmsg('Aquired: %s' % os.getpid())
    def logUnlock(self):
        self.logmsg('Released: %s' % os.getpid())
```

Этот класс управления блокировками файлов намеренно запрограммирован в собственном модуле, так как его можно будет повторно использовать. В PyErrata классы баз данных полок встраивают его с помощью *множественного наследования*, чтобы реализовать взаимное исключение для процессов записи в базу данных.

Этот класс предполагает существование блокируемого файла с именем `self.filename` (определяемым в классах клиентов) с расширением *.lck*; как и все атрибуты экземпляра, это имя может быть различным для каждого клиента класса. Если соответствующая глобальная переменная имеет значение «истина», этот класс также регистрирует все операции блокировки в файле с тем же именем, что у блокируемого, но с расширением *.log*.

Обратите внимание на то, что файл журнала открывается в режиме дописывания `a`; в Unix-системах этот режим гарантирует, что текст, записываемый каждым процессом, появляется на собственной строчке, не перемешиваясь (несколько копий этого класса могут осуществлять запись в журнал из параллельных процессов сценариев CGI). Однако чтобы действительно понять, как работает этот класс, необходимо остановиться на интерфейсе Python для блокировки файлов.

Блокировка файлов с помощью fcntl.flock

Когда мы изучали потоки в главе 3 «Системные средства параллельного выполнения», то видели, что модуль Python thread содержит механизм блокировки со взаимным исключением, который может быть использован для синхронизации доступа потоков к совместно используемым ресурсам глобальной памяти. Однако в среде CGI от этого обычно мало пользы, потому что каждый запрос к базе данных обычно поступает из отдельного процесса, порожденного сервером HTTP для обработки поступившего запроса. То есть блокировки потоков действуют только внутри одного и того же процесса, потому что все потоки выполняются в *одном* процессе.

Для сценариев же CGI нам обычно требуется механизм блокировки, простирающийся на несколько процессов. В системах Unix стандартная библиотека Python экспортирует инструмент, основанный на блокировке файлов, и потому доступный через границы процессов. Все чудеса происходят в следующих двух строках класса мьютекса PyErrata:

```
fcntl.flock(self.lock.fileno(), mode)       # ждет блокировки при необходимости
fcntl.flock(self.lock.fileno(), LOCK_UN)    # разблокировать для других процессов
```

Вызов `fcntl.flock` в стандартной библиотеке Python пытается захватить блокировку, ассоциированную с файлом, и по умолчанию при необходимости приостанавливает вызвавший процесс, пока не будет получена блокировка. Вызов принимает целочисленный код *дескриптора файла* (его возвращает метод `fileno` объекта файла `stdio`) и *флаг режима*, определенный в стандартном модуле `FCNTL`, который в нашей системе может иметь одно из трех значений:

- `LOCK_EX` запрашивает монопольную блокировку, обычно используемую в процессах записи. Эта блокировка предоставляется только при отсутствии других блокировок (монопольных или совместных) и приостанавливает все другие запросы блокировок (монопольных или совместных), пока блокировка не будет снята. Это гарантирует,что держатель монопольной блокировки выполняется без помех.
- `LOCK_SH` запрашивает совместно используемую блокировку, обычно используемую процессами чтения. Одновременно удерживать совместные блокировки может любое количество процессов, но блокировка предоставляется, только если нет монопольной блокировки, а новые запросы монопольных блокировок приостанавливаются, пока не будут сняты все совместные блокировки.
- `LOCK_UN` снимает блокировку, ранее захваченную вызывающим процессом, чтобы другие процессы могли захватить блокировку и возобновить выполнение.

На языке баз данных итоговый результат выражается в том, что читающие процессы должны ждать, только если другой процесс держит блокировку по записи, а пишущие процессы ждут, пока удерживается любая блокировка – по чтению или записи. Используемая для синхронизации процессов, эта схема более сложная и мощная, чем простая модель захвата/освобождения блокировок модуля Python `thread`, и отличается от инструментов классов, имеющихся в модуле более высокого уровня `threading`. Однако эта схема может эмулироваться обоими этими модулями потоков.

`fcntl.flock` внутренне обращается к тому механизму блокировки файлов, который есть в базовой операционной системе,[1] поэтому за дополнительными подробностями можно обратиться к соответствующей странице руководства по Unix или Linux. Есть также возможность избежать приостановки при невозможности получить блокировку, и существуют другие средства синхронизации в библиотеке Python (например, очереди – «fifo»), но здесь мы не будем касаться этих вариантов.

Тестовые сценарии для мьютексов

Чтобы легче было понять модель синхронизации PyErrata, выполним несколько простых экспериментов для получения лучшего представления о базовых элементах блокировки файлов. В примерах 14.18 и 14.19 реализованы простые процессы чтения и записи, непосредственно вызывающие `flock` без обращения к нашему классу. Они запрашивают совместную и монопольную блокировки соответственно.

Пример 14.18. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testread.py

```
#!/usr/bin/python

import os, fcntl, time
```

[1] Механизмы блокировки зависят от платформы и могут остутствовать вообще. Например, вызов `flock` в настоящее время не поддерживается в Windows для Python 1.5.2, поэтому на некоторых серверах может потребоваться заменить этот вызов на альтернативный для данной платформы.

```
from FCNTL import LOCK_SH, LOCK_UN
print os.getpid(), 'start reader', time.time()

file = open('test.lck', 'r')              # открыть файл блокировки
fcntl.flock(file.fileno(), LOCK_SH)       # приостановка, если блокировка у пишущего процесса
print os.getpid(), 'got read lock', time.time()    # может выполняться любое
                                                   # число читающих процессов
time.sleep(3)
print 'lines so far:', os.popen('wc -l Shared.txt').read(),

print os.getpid(), 'unlocking\n'
fcntl.flock(file.fileno(), LOCK_UN)  # возобновить приостановленные процессы записи
```

В этом простом тесте для синхронизации чтения и записи в текстовый файл, в который процессы записи осуществляют дописывание, используется блокировка текстового файла *test.lck*. Текстовый файл, к которому дописываются данные, играет роль полочных баз данных PyErrata, а сценарии чтения и записи примеров 14.18 и 14.19 замещают процессы сценариев просмотра и передачи.

Пример 14.19. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testwrite.py

```
#!/usr/bin/python

import os, fcntl, time
from FCNTL import LOCK_EX, LOCK_UN
print os.getpid(), 'start writer', time.time()

file = open('test.lck', 'r')                       # открыть файл блокировки
fcntl.flock(file.fileno(), LOCK_EX)                # приостановка, если есть чтение или запись
print os.getpid(), 'got write lock', time.time()  # только 1 пишущий процесс в каждый момент
log = open('Shared.txt', 'a')
time.sleep(3)
log.write('%d Hello\n' % os.getpid())

print os.getpid(), 'unlocking\n'
fcntl.flock(file.fileno(), LOCK_UN)  # возобновить приостановленные процессы чтения или записи
```

Чтобы начать параллельную работу нескольких процессов чтения и записи, в примере 14.20 используется комбинация вызовов Unix `fork/execl`, которая запускает программные процессы (оба вызова описаны в главе 3).

Пример 14.20. PP2E\Internet\Cgi-Web\PyErrata\Mutex\launch-test.py

```
#!/usr/bin/python
########################################################################################################
# Запуск тестовых программных процессов как ./launch-test.py > launch-test.out
# попробуйте порождать чтение перед записью, затем запись перед чтением – в обоих случаях
# второй процесс приостанавливается до снятия блокировки первым; если сначала запускаются
# 2 процесса чтения, оба получают блокировку и приостанавливают пишущий процесс;
# если сначала запускаются 2 процесса записи, а затем 2 чтения, 2-й пишущий ждет первого,
# оба чтения ждут обоих процессов записи, и оба чтения получают блокировку одновременно;
# в тесте ниже выполняется первый пишущий, затем все читающие перед любым пишущим;
# если вначале читающие процессы, все выполняются перед любым пишущим (все под linux)
########################################################################################################

import os

for i in range(1):
    if os.fork() == 0:
        os.execl("./testwrite.py")

for i in range(2):                        # копировать этот процесс
```

```
    if os.fork() == 0:                  # если в новом дочернем процессе,
        os.execl("./testread.py")       # запуск тестовой программы

for i in range(2):
    if os.fork() == 0:
        os.execl("./testwrite.py")      # то же, но запуск записи

for i in range(2):
    if os.fork() == 0:
        os.execl("./testread.py")

for i in range(1):
    if os.fork() == 0:
        os.execl("./testwrite.py")
```

Комментарии в начале этого сценария описывают результаты выполнения его логики под Linux при различных условиях. Практическое замечание: после копирования этих файлов с Windows по FTP в файле `tar` мне пришлось дать им разрешение на выполнение и преобразовать переводы строки из формата DOS в формат Unix, чтобы Linux обращалась с ними как с выполняемыми программами:[1]

```
[mark@toy .../PyErrata/Mutex]$ chmod +x *.py
[mark@toy .../PyErrata/Mutex]$ python $X/PyTools/fixeoln_all.py tounix "*.py"
__init__.py
launch-mutex-simple.py
launch-mutex.py
launch-test.py
mutexcntl.py
testread-mutex.py
testread.py
testwrite-mutex.py
testwrite.py
```

После настройки как выполняемых программ можно запустить все три эти сценария из командной строки Linux. Сценарии чтения и записи обращаются к файлу *Shared.txt*, который имитирует разделяемый ресурс в реальном параллельном приложении (например, базу данных, используемую в CGI):

```
[mark@toy ...PyErrata/Mutex]$ ./testwrite.py
1010 start writer 960919842.773
1010 got write lock 960919842.78
1010 unlocking

[mark@toy ...PyErrata/Mutex]$ ./testread.py
1013 start reader 960919900.146
1013 got read lock 960919900.153
lines so far:     132 Shared.txt
1013 unlocking
```

Сценарий `launch-test` просто запускает пакет сценариев чтения и записи, выполняющихся как параллельные процессы, чтобы моделировать среду с одновременными об-

---

[1] Синтаксис `+x` в команде оболочки *chmod* означает «установить бит выполнения» в битовой строке прав доступа к файлу для «себя», текущего пользователя. На моей машине, во всяком случае, *chmod* принимает как целочисленные битовые строки, как раньше, так и этот символический формат. Обратите внимание, что эти тесты выполняются в Linux, потому что вызов Python `os.fork` не работает в Windows, по крайней мере, для Python 1.5.2. Когда-то он, возможно, будет работать, но в сценариях Windows вместо него используется `os.spawnv` (подробности в главе 3).

ращениями (например, одновременная связь со сценарием CGI нескольких веб-браузеров):

```
[mark@toy ...PyErrata/Mutex]$ python launch-test.py
1016 start writer 960919933.206
1016 got write lock 960919933.213
1017 start reader 960919933.416
1018 start reader 960919933.455
1022 start reader 960919933.474
1021 start reader 960919933.486
1020 start writer 960919933.497
1019 start writer 960919933.508
1023 start writer 960919933.52
1016 unlocking

1017 got read lock 960919936.228
1018 got read lock 960919936.234
1021 got read lock 960919936.24
1022 got read lock 960919936.246
lines so far:     133 Shared.txt
1022 unlocking

lines so far:     133 Shared.txt
1018 unlocking

lines so far:     133 Shared.txt
1017 unlocking

lines so far:     133 Shared.txt
1021 unlocking

1019 got write lock 960919939.375
1019 unlocking

1020 got write lock 960919942.379
1020 unlocking

1023 got write lock 960919945.388
1023 unlocking
```

Этот вывод несколько загадочен; в большинстве строк выводятся ID процесса, текст и системное время, а каждый процесс вставляет трехсекундную задержку (через `time.sleep`), моделирующую реальную занятость. Если посмотреть внимательно, то можно заметить, что все процессы начинаются примерно в одно время, но доступ к совместно используемому файлу синхронизируется в следующей последовательности:

1. Сначала один пишущий процесс захватывает файл.
2. Затем его в одно время получают все читающие процессы, через три секунды.
3. Наконец, все другие пишущие процессы получают файл один за другим, с интервалом три секунды.

В итоге всегда один процесс записи получает доступ к файлу, а остальные процессы в это время приостанавливаются. При такой последовательности проблем с одновременным обновлением не возникает.

Сценарии тестирования класса mutex

Чтобы проверить наш класс мьютекса вне PyErrata, перепишем эти сценарии, чтобы они обращались к интерфейсу класса. Вывод примеров 14.21 и 14.22 аналогичен предыдущим версиям с простым `fcntl`, но появляется дополнительный журнальный файл для облегчения трассировки операций блокирования.

Пример 14.21. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testread-mutex.py

```
#!/usr/bin/python
import os, time
from mutexcntl import MutexCntl

class app(MutexCntl):
    def go(self):
        self.filename = 'test'
        print os.getpid(), 'start mutex reader'
        self.sharedAction(self.report)          # можно report вместе с другими,
                                                # но не во время обновления
    def report(self):
        print os.getpid(), 'got read lock'
        time.sleep(3)
        print 'lines so far:', os.popen('wc -l Shared.txt').read(),
        print os.getpid(), 'unlocking\n'

if __name__ == '__main__': app().go()
```

В отличие от PyErrata, в примерах 14.21 и 14.22 не требуется изменять `sys.path`**, чтобы позволить импортировать из** `FCNTL` **в модуле** `mutexcntl`**, потому что мы будем выполнять эти сценарии под своим именем, а не под именем «nobody» пользователя CGI (в моем пути есть каталог, в котором находится** `FCNTL`**).**

Пример 14.22. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testwrite-mutex.py

```
#!/usr/bin/python
import os, time
from mutexcntl import MutexCntl

class app(MutexCntl):
    def go(self):
        self.filename = 'test'
        print os.getpid(), 'start mutex writer'
        self.exclusiveAction(self.update)       # должен делать это один;
                                                # обновление или отчет нельзя
    def update(self):                           # выполнять в одно время
        print os.getpid(), 'got write lock'
        log = open('Shared.txt', 'a')
        time.sleep(3)
        log.write('%d Hello\n' % os.getpid())
        print os.getpid(), 'unlocking\n'

if __name__ == '__main__': app().go()
```

Программа запуска та же, что в примере 14.20, но пример 14.23 запускает несколько экземпляров основанных на классе процессов чтения и записи. Запустите пример 14.23 на своем сервере с разными значениями счетчика процессов, чтобы понаблюдать за механизмом блокировки.

Пример 14.23. PP2E\Internet\Cgi-Web\PyErrata\launch-mutex.py

```
#!/usr/bin/python
# запуск процессов тестирующей программы
# то же, но запускаются клиенты mutexcntl

import os

for i in range(1):
    if os.fork() == 0:
```

```
        os.execl("./testwrite-mutex.py")

for i in range(2):
    if os.fork() == 0:
        os.execl("./testread-mutex.py")

for i in range(2):
    if os.fork() == 0:
        os.execl("./testwrite-mutex.py")

for i in range(2):
    if os.fork() == 0:
        os.execl("./testread-mutex.py")

for i in range(1):
    if os.fork() == 0:
        os.execl("./testwrite-mutex.py")
```

Вывод теста, использующего классы, примерно тот же, что и раньше. Процессы запускаются в другом порядке, но синхронизация происходит идентично – один процесс пишет, все читающие читают, затем остальные пишущие процессы поочередно выполняют запись:

```
[mark@toy .../PyErrata/Mutex]$ python launch-mutex.py
1035 start mutex writer
1035 got write lock
1037 start mutex reader
1040 start mutex reader
1038 start mutex writer
1041 start mutex reader
1039 start mutex writer
1036 start mutex reader
1042 start mutex writer
1035 unlocking

1037 got read lock
1041 got read lock
1040 got read lock
1036 got read lock
lines so far:     137 Shared.txt
1036 unlocking

lines so far:     137 Shared.txt
1041 unlocking

lines so far:     137 Shared.txt
1040 unlocking

lines so far:     137 Shared.txt
1037 unlocking

1038 got write lock
1038 unlocking

1039 got write lock
1039 unlocking

1042 got write lock
1042 unlocking
```

На этот раз все временные метки удалены из вывода программы запуска, потому что наш класс мьютекса автоматически регистрирует операции блокировки в отдельном

файле вместе с временем и ID процессов; трехсекундная задержка в каждом процессе более очевидна в таком формате:

```
[mark@toy .../PyErrata/Mutex]$ cat test.log
960920109.518   Requested: 1035, writer
960920109.518   Aquired: 1035
960920109.626   Requested: 1040, reader
960920109.646   Requested: 1038, writer
960920109.647   Requested: 1037, reader
960920109.661   Requested: 1041, reader
960920109.674   Requested: 1039, writer
960920109.69    Requested: 1036, reader
960920109.701   Requested: 1042, writer
960920112.535   Released: 1035
960920112.542   Aquired: 1037
960920112.55    Aquired: 1041
960920112.557   Aquired: 1040
960920112.564   Aquired: 1036
960920115.601   Released: 1036
960920115.63    Released: 1041
960920115.657   Released: 1040
960920115.681   Released: 1037
960920115.681   Aquired: 1038
960920118.689   Released: 1038
960920118.696   Aquired: 1039
960920121.709   Released: 1039
960920121.716   Aquired: 1042
960920124.728   Released: 1042
```

Наконец, вот как выглядит совместно используемый текстовый файл после завершения всех этих процессов. Каждый пишущий процесс просто добавил строку со своим ID процесса; это не самый впечатляющий из результатов параллельной обработки, но если представить себе, что это использующая shelve *база данных* нашей системы PyErrata, то эти проверки окажутся весьма многозначительными:

```
[mark@toy .../PyErrata/Mutex]$ cat Shared.txt
1010 Hello
1016 Hello
1019 Hello
1020 Hello
1023 Hello
1035 Hello
1038 Hello
1039 Hello
1042 Hello
```

Средства администрирования

Теперь, когда мы закончили реализацию на Python размещенной в веб базы данных сообщений с возможностью одновременного доступа и опубликовали веб-страницы и сценарии, делающие эту базу данных доступной во всем киберпространстве, можно спокойно ждать сообщений, которые поступят. Или почти так: у владельца сайта пока нет способа просмотра или удаления записей в автономном режиме. Более того, все записи при передаче помечаются как «not yet verified» (не проверено), и их нужно каким-то образом проверять или отклонять.

В этом разделе приведен ряд сжато документированных сценариев PyErrata, решающих такие задачи. Все они являются программами Python, помещаемыми в каталог верхнего уровня *AdminTools*, и могут запускаться из командной строки оболочки на сервере (или другой машине, если скачать туда базу данных). В них реализованы простой дамп содержимого базы данных, резервное копирование ее и изменение состояния базы данных и удаление записей, что осуществляет администратор сайта.

Эти задачи выполняются не очень часто, поэтому на создание данных инструментов не было затрачено много усилий. Честно говоря, некоторые из них относятся к категории сляпанных наспех («quick and dirty») и не столь надежны, как хотелось бы. Например, поскольку эти сценарии обходят классы интерфейсов базы данных и непосредственно обращаются к базовым структурам файлов, изменения в используемых файловых механизмах могут привести их в негодность. Кроме того, в более совершенных будущих версиях из этих инструментов могут вырасти интерфейсы пользователя, основанные на GUI или веб и поддерживающие удаленное администрирование через сеть. В настоящий момент такие расширения оставляются в качестве упражнений честолюбивым читателям.

Средства создания резервных копий

Системные средства создания резервных копий просто порождают стандартные программы командной строки Unix `tar` и `gzip`, чтобы скопировать базу данных в один сжатый файл. Можно написать для этой цели сценарий оболочки, но с таким же успехом работает Python, как показывают примеры 14.24 и 14.25.

Пример 14.24. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\backupFiles.py

```
#!/usr/bin/python
import os
os.system('tar -cvf DbaseFiles.tar ../DbaseFiles')
os.system('gzip DbaseFiles.tar')
```

Пример 14.25. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\backupShelve.py

```
#!/usr/bin/python
import os
os.system('tar -cvf DbaseShelve.tar ../DbaseShelve')
os.system('gzip DbaseShelve.tar')
```

Средства отображения

Сценарии примеров 14.26 и 14.27 создают необработанные дампы содержимого каждой базы данных. Поскольку базы данных используют механизмы хранения на чистом Python (сериализованные файлы, полки), эти сценарии могут работать на один уровень ниже, чем опубликованные классы интерфейсов баз данных; должны ли они это делать, зависит от того, какой объем кода вы готовы поменять, когда ваша модель базы данных будет развиваться. Кроме вывода генерируемых имен файлов записей и ключей полок, нет причин не сделать эти сценарии менее хрупкими, вызывая вместо этого методы `loadSortedTable` классов баз данных. Упражнение для читателя: сделайте лучше.

Пример 14.26. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\dumpFiles.py

```
#!/usr/bin/python
import glob, pickle

def dump(kind):
    print '\n', kind, '='*60, '\n'
```

```
    for file in glob.glob("../DbaseFiles/%s/*.data" % kind):
        print '\n', '-'*60
        print file
        print pickle.load(open(file, 'r'))

dump('errataDB')
dump('commentDB')
```

Пример 14.27. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\dumpShelve.py

```
#!/usr/bin/python
import shelve
e = shelve.open('../DbaseShelve/errataDB')
c = shelve.open('../DbaseShelve/commentDB')

print '\n', 'Errata', '='*60, '\n'
print e.keys()
for k in e.keys(): print '\n', k, '-'*60, '\n', e[k]

print '\n', 'Comments', '='*60, '\n'
print c.keys()
for k in c.keys(): print '\n', k, '-'*60, '\n', c[k]
```

При выполнении этих сценариев получаются результаты следующего вида (усечены до 80 символов, чтобы уместиться на странице). Они далеко не так красивы, как веб-страницы, генерируемые для пользователя в PyErrata, но их можно направлять в другие сценарии командной строки для дальнейшего автономного анализа и обработки. Например, вывод сценария дампа можно отправить сценарию генерации отчета, не имеющему понятия о веб:

```
[mark@toy .../Internet/Cgi-Web/PyErrata/AdminTools]$ python dumpFiles.py

errataDB ============================================================

------------------------------------------------------------
../DbaseFiles/errataDB/937907956.159-5157.data
{'Page number': '42', 'Type': 'Typo', 'Severity': 'Low', 'Chapter number': '3'...

------------------------------------------------------------
...more...

commentDB ===========================================================

------------------------------------------------------------
../DbaseFiles/commentDB/937908410.203-5352.data
{'Submit date': '1999/09/21, 06:06:50', 'Submitter email': 'bob@bob.com',...

------------------------------------------------------------
...more...

[mark@toy .../Internet/Cgi-Web/PyErrata/AdminTools]$ python dumpShelve.py

Errata ==============================================================

['938245136.363-20046', '938244808.434-19964']

938245136.363-20046 ------------------------------------------------------------
{'Page number': '256', 'Type': 'Program bug', 'Severity': 'High', 'Chapter nu...

938244808.434-19964 ------------------------------------------------------------
{'Page number': 'various', 'Type': 'Suggestion', 'Printing Date': '', 'Chapte...

Comments ============================================================
```

```
['938245187.696-20054']

938245187.696-20054 ------------------------------------------------------------
{'Submit date': '1999/09/25, 03:39:47', 'Submitter email': 'bob@bob.com', 'Re...
```

Средства изменения статуса сообщения

Наша последняя группа инструментов командной строки позволяет владельцу сайта помечать сообщения как проверенные или отклоненные, а также вообще удалять сообщения. Мысль в том, что кто-нибудь будет иногда запускать эти сценарии а автономном режиме, когда у него есть время, и изменять статус сообщения после его изучения. Этим завершается наша задача автоматизации сбора сообщений об ошибках: сам процесс изучения предполагает наличие времени и разума.

В классах баз данных нет интерфейсов для изменения имеющихся сообщений, поэтому данные сценарии могут, по крайней мере, создать прецедент для того, чтобы опуститься дальше классов, к средствам физического хранения. С другой стороны, можно было бы расширить классы, чтобы они поддерживали такие операции обновления тоже, с помощью интерфейсов, которые могли бы также использоваться в будущем средствами изменения состояния (например, веб-интерфейсов).

Чтобы несколько уменьшить избыточность, сначала определим функции изменения состояния в общем модуле, приведенном в примере 14.28, чтобы их можно было совместно использовать в сценариях, основанных на файлах и на полках.

Пример 14.28. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\verifycommon.py

```
##############################################################################
# Поместить общий код проверки в отдельный модуль для единообразия и повторного
# использования; можно было бы обобщить обновление базы данных, но достаточно и этого
##############################################################################

def markAsVerify(report):
    report['Report state'] = 'Verified by author'

def markAsReject(report):
    reason = ''                                 # ввести текст причины отклонения
    while 1:                                    # предпослать исходному описанию
        try:
            line = raw_input('reason>')
        except EOFError:
            break
        reason = reason + line + '\n'
    report['Report state'] = 'Rejected - not a real bug'
    report['Description']  = ('Reject reason: ' + reason +
                '\n[Original description=>]\n' + report['Description'])
```

Для изменения состояний в базе, основанной на файлах, просто осуществляется обход всех сериализованных файлов в каталогах баз данных, как показано в примере 14.29.

Пример 14.29. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\verifyFiles.py

```
#!/usr/bin/python
#############################################################################################
# Операции изменения состояния и удаления сообщений; требуется также средство публикации
# анонимных сообщений, отправленных по почте и представляющих большой интерес - пока они
# вводятся с помощью форм передачи вручную; используется текстовый режим: смысл в том, чтобы
# сначала просмотреть записи на странице ошибок (отсортированные по состоянию, непроверенные -
# в начале), но было бы очень полезно добавить интерфейс проверки, основанный на gui или веб;
#############################################################################################
```

```
import glob, pickle, os
from verifycommon import markAsVerify, markAsReject
def analyse(kind):
    for file in glob.glob("../DbaseFiles/%s/*.data" % kind):
        data = pickle.load(open(file, 'r'))
        if data['Report state'] == 'Not yet verified':
             print data
             if raw_input('Verify?') == 'y':
                 markAsVerify(data)
                 pickle.dump(data, open(file, 'w'))
             elif raw_input('Reject?') == 'y':
                 markAsReject(data)
                 pickle.dump(data, open(file, 'w'))
             elif raw_input('Delete?') == 'y':
                 os.remove(file)  # то же, что os.unlink
print 'Errata...';   analyse('erratDB')
print 'Comments...'; analyse('commentDB')
```

Этот сценарий выполняется из командной строки и поочередно выводит содержимое сообщений, спрашивая после каждого, нужно ли его подтвердить, отклонить или удалить. Вот начало одного сеанса проверки файловой базы данных, приведенное с переносом строк, чтобы показать то же, что вижу я (оно неровное, но компактное):

```
[mark@toy .../Internet/Cgi-Web/PyErrata/AdminTools]$ python verifyFiles.py
Errata...
{'Page number': '12', 'Type': 'Program bug', 'Printing Date': '', 'Chapter numbe
r': '', 'Submit date': '1999/09/21, 06:17:13', 'Report state': 'Not yet verified
', 'Submitter name': 'Lisa Lutz', 'Submitter email': '', 'Description': '1 + 1 =
 2, not 3...\015\012', 'Submit mode': '', 'Part number': '', 'Severity
': 'High'}
Verify?n
Reject?n
Delete?n
{'Page number': '', 'Type': 'Program bug', 'Printing Date': '', 'Chapter number'
: '16', 'Submit date': '1999/09/21, 06:20:22', 'Report state': 'Not yet verified
', 'Submitter name': 'jerry', 'Submitter email': 'http://www.jerry.com', 'Descri
ption': 'Help! I just spilled coffee all over my\015\012computer...\015\012
     ', 'Submit mode': '', 'Part number': '', 'Severity': 'Unknown'}
Verify?n
Reject?y
reason>It's not Python's fault
reason>(ctrl-d)
...еще строки...
```

Подтверждения и отклонения изменяют записи, но при удалении они действительно изымаются из системы. В `verifycommon` при отклонении сообщения предлагается ввести объяснение, которое добавляется к первоначальному описанию. При удалении файл сообщения удаляется вызовом `os.remove`; эта функция может оказаться удобной, если системой воспользуется несерьезный пользователь (включая меня при составлении примеров для этой книги). Версия этого сценария, основанная на полках, выглядит аналогично, но действует с полками, а не с плоскими файлами, как показано в примере 14.30.

Пример 14.30. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\verifyShelve.py

```
#!/usr/bin/python
##########################################################
# Действует аналогично verifyFiles.py, но с файлами shelve;
```

```
# предупреждение: перед обновлением нужно осуществить блокировку,
# и есть некоторая избыточность кода в просмотре базы данных
#########################################################

import shelve
from verifycommon import markAsVerify, markAsReject

def analyse(dbase):
    for k in dbase.keys():
        data = dbase[k]
        if data['Report state'] == 'Not yet verified':
             print data
             if raw_input('Verify?') == 'y':
                 markAsVerify(data)
                 dbase[k] = data
             elif raw_input('Reject?') == 'y':
                 markAsReject(data)
                 dbase[k] = data
             elif raw_input('Delete?') == 'y':
                 del dbase[k]

print 'Errata...';    analyse(shelve.open('../DbaseShelve/erratDB'))
print 'Comments...';  analyse(shelve.open('../DbaseShelve/commentDB'))
```

Обратите внимание, что модуль `verifycommon` обеспечивает единообразную маркировку записей и частично устраняет избыточность. Однако сценарии проверки для файлов и полок все же сохраняют большое сходство; можно было бы попытаться обобщить идею просмотра базы данных для обновления, переместив эту логику в показанные ранее классы специфических для баз данных интерфейсов.

За этим исключением, с избыточностью логики просмотра или зависимостью от структуры сценариев проверки, основанных на файлах и полках, ничего особенно не сделать. Имеющиеся методы загрузки списка класса базы данных бесполезны, так как не предоставляют сгенерированные имя файла и ключ полки, необходимые для переписывания записи. Чтобы сделать средства администрирования более надежными, нужны, вероятно, некоторые изменения в конструкции класса базы данных, и это неплохой переход к следующему разделу.

Проектирование с учетом повторного использования и расширения

Я признаю, что PyErrata, возможно, экономна, но она также несколько эгоистична. Интерфейсы баз данных, представленные в предыдущих разделах, действуют как задумано и служат отделению всей обработки баз данных от деталей сценариев CGI. Но как показано в книге, эти интерфейсы могли бы иметь более широкие возможности повторного использования; кроме того, они не предназначены для масштабирования и работы с крупными приложениями баз данных.

Завершим эту главу кратким обзором нашего программного кода и рассмотрением некоторых альтернативных архитектур PyErrata. В этом разделе я подчеркиваю некоторые особенности интерфейса баз данных PyErrata, препятствующие широте их применения, не для умаления собственного достоинства, а чтобы показать, как программистские решения могут влиять на повторное использование кода.

Этот раздел касается и других вопросов. Здесь в большей степени представлены общие идеи, нежели код, а тот код, который есть, скорее представляет собой экспериментальное проектирование, а не конечный продукт. С другой стороны, поскольку

этот проект пишется на Python, его можно запускать для проверки осуществимости альтернативных проектов; как мы уже видели, Python можно использовать как род *исполняемого псевдокода*.

Возможность повторного использования

Как мы видели, повторное использование кода всюду присутствует в PyErrata: вызовы верхнего уровня проходят в общие модули просмотра и передачи, которые, в свою очередь, вызывают классы баз данных, повторно использующие общий модуль. Но можно ли использовать код PyErrata в других системах? Модули интерфейсов баз данных PyErrata не проектировались с учетом общности, но они *почти* пригодны для реализации других видов баз данных, основанных на файлах и полках, вне контекста самой PyErrata. Однако для того чтобы превратить эти интерфейсы в инструменты, которые можно широко использовать, требуется некоторая дополнительная модификация.

В настоящем виде имена каталогов полок и файлов жестко закодированы в модулях, специфических для способа хранения подклассов, но другая система может импортировать и повторно использовать их классы `Dbase`, задав другие имена каталогов. Однако, уменьшая общность, модуль `dbcommon` добавляет во все новые записи два атрибута (время передачи и состояние сообщения), что может быть неуместно за рамками PyErrata. Он также предполагает, что записываемые значения являются отображениями (словарями), но это менее связано со спецификой PyErrata.

Если переписывать эти классы в более обобщенном виде, то есть смысл переоформить четыре класса `DbaseErrata` и `DbaseComment` в виде самостоятельных модулей (это весьма специфические случаи баз данных файлов и полок). Вероятно, потребовалось бы также переместить вставку атрибутов времени передачи и состояния сообщения из модуля `dbcommon` непосредственно в эти четыре класса (эти атрибуты специфичны для баз данных PyErrata). Например, можно определить новый класс `DbasePyErrata`, устанавливающий эти атрибуты и являющийся суперклассом для четырех специфичных для способа хранения классов баз данных PyErrata:

```
# в новом модуле
class DbasePyErrata:
     def storeItem(self, newdata):
        secsSinceEpoch          = time.time()
        timeTuple               = time.localtime(secsSinceEpoch)
        y_m_d_h_m_s             = timeTuple[:6]
        newdata['Submit date']  = '%s/%02d/%02d, %02d:%02d:%02d' % y_m_d_h_m_s
        newdata['Report state'] = 'Not yet verified'
        self.writeItem(newdata)

# в dbshelve
class Dbase(MutexCntl, dbcommon.Dbase):
    # как есть

# в dbfiles
class Dbase(dbcommon.Dbase):
    # как есть

# в новом модуле файлов
class DbaseErrata(DbasePyErrata,  dbfiles.Dbase):
    dirname = 'DbaseFiles/errataDB/'
class DbaseComment(DbasePyErrata, dbfiles.Dbase):
    dirname = 'DbaseFiles/commentDB/'
```

```
# в новом модуле полок
class DbaseErrata(DbasePyErrata,  dbshelve.Dbase):
    filename = 'DbaseShelve/errataDB'
class DbaseComment(DbasePyErrata, dbshelve.Dbase):
    filename = 'DbaseShelve/commentDB'
```

У нас не хватит здесь места, чтобы осветить все возможные структуры. Смысл в том, что после вычленения кода, специфичного для приложения, модули `dbshelve` и `dbfiles` не только позволяют отчетливее разделить интерфейс PyErrata и код базы данных, но и становятся средством хранения данных с возможностью широкого использования.

Масштабируемость

Интерфейсы баз данных PyErrata были разработаны лишь для задач хранения данных этого конкретного приложения и не годятся для очень больших баз данных. Если внимательно изучить код баз данных, то можно заметить, что операции передачи обновляют один элемент, но запрос на просмотр целиком загружает в память базы данных сообщений. Такая схема удовлетворяет тем размерам баз данных, которые ожидаются в PyErrata, но плохо действует с большими наборами данных. Можно расширить классы баз данных, чтобы они смогли работать и с большими наборами данных, но при этом, видимо, потребуются также новые интерфейсы верхнего уровня.

Когда я прекратил редактировать статический файл HTML со списком сообщений об ошибках для первого издания этой книги, в нем было около 60 сообщений, и я рассчитываю на такой же малый объем данных для других книг и изданий. Когда базы данных такие маленькие, приемлемо загружать базы данных в память целиком (то есть в списки и словари Python), причем часто. Действительно, время, необходимое для передачи веб-страницы с 60 записями через Интернет, вероятно, перевесит время загрузки 60 файлов сообщений или ключей полок на сервере.

С другой стороны, работа базы данных может очень замедлиться, если будет передано намного больше сообщений, чем ожидается. Существенно оптимизировать варианты вывода «Simple list» и «With index» не удастся, так как они действительно выводят все записи. Но для варианта «Index only» можно изменить наши классы так, чтобы загружались только записи с выбранным значением в назначенном поле сообщения.

Например, можно преодолеть узкие места загрузки базы данных, изменив наши классы так, чтобы они реализовывали *отсроченную загрузку* записей: вместо возврата реальной базы данных запросы загрузки могли бы возвращать объекты, выглядящие так же, но получающие действительные записи только при необходимости. При таком подходе может не потребоваться изменять остальную часть кода системы, но реализовать его может быть труднее.

Индексирование по нескольким полям в файле shelve

Лучшим подходом, вероятно, будет определение совершенно нового интерфейса верхнего уровня для варианта «Index only», при котором действительно загружаются только записи, соответствующие запрашиваемому значению поля. Например, вместо того чтобы хранить все записи на одной полке, можно реализовать базу данных как *набор индексных полок*, по одной для каждого поля записи, чтобы связать записи по значениям полей. *Ключами* индексных полок будут значения связанного поля; *значениями* полок будут списки записей с одинаковым значением поля. Списки элементов полок могут содержать избыточные экземпляры записей или уникальные имена плоских файлов с сериализованными словарями записей, внешние по отношению к индексным полкам (как в текущей модели плоских файлов).

Например, база данных комментариев PyErrata может быть организована как *каталог плоских файлов*, содержащих сериализованные словари сообщений, а также пять полок для индексации значений во всех полях записей (имя передавшего, его адрес электронной почты, режиме передачи, дата передачи, состояние сообщения). На полке состояния сообщения должен быть один элемент для каждого возможного состояния сообщения (подтверждено, отклонено и т. д.); каждый элемент будет содержать список записей, у которых значение состояния сообщения соответствует этому элементу. Запросы по значению поля будут выполняться быстро, но операции записи и загрузки станут более сложными:

- Чтобы сохранить запись в такой схеме, мы должны сначала сериализовать ее в плоский файл с уникальным именем, затем поместить имя этого файла в списки на всех пяти полках, используя в качестве ключа полки значение каждого поля.
- Чтобы загрузить только те записи, которые соответствуют комбинации поле/значение, нужно сначала получить список имен файлов по индексу значения на полке этого поля, а затем пройти этот список и загрузить только совпавшие записи из плоских файлов сериализации.

Перейдем сразу от гипотез к конкретным действиям и создадим на Python прототип этих идей. Если вы внимательно следите за изложением, то заметите, что в действительности мы говорим о *расширении* структуры базы данных на плоских файлах, в котором просто добавляются индексные полки. Поэтому одним из возможных способов реализации этой модели является создание подклассов текущих классов плоских файлов. В примере 14.31 это и делается, как проверка концепции проекта.

Пример 14.31. PP2E\Internet\PyErrata\AdminTools\dbaseindexed.py

```
############################################################################################
# Добавить полки индексов полей в механизм баз данных на плоских файлах;
# для оптимизации вывода "index only" используйте классы в конце этого файла;
# изменить просмотр, индекс, передачу, чтобы использовать новые загрузчики в режиме
# "Index only"; мелочь: использует один файл блокировки для всех операций чтения/записи
# индексных полок; хранение копий записей вместо имен файлов на индексных полках несколько
# ускорило бы работу (избегая открытия плоских файлов), но заняло бы больше места;
# возвращается к первоначальной логике грубой загрузки для неиндексированных полей;
# shelve.open создает пустой файл, если его еще не существует, поэтому отказов не бывает;
# для запуска создайте DbaseFilesIndex/{commentDB,erratasDB}/indexes.lck;
############################################################################################

import sys; sys.path.insert(0, '..')                # сначала проверить родительский каталог
                                                    # для администрации
from Mutex import mutexcntl                         # годится путь fcntl: не 'nobody'
import dbfiles, shelve, pickle, string, sys

class Dbase(mutexcntl.MutexCntl, dbfiles.Dbase):
    def makeKey(self):
        return self.cachedKey
    def cacheKey(self):                             # сохранить имя файла
        self.cachedKey = dbfiles.Dbase.makeKey(self) # оно здесь тоже нужно
        return self.cachedKey

    def indexName(self, fieldname):
        return self.dirname + string.replace(fieldname, ' ', '-')

    def safeWriteIndex(self, fieldname, newdata, recfilename):
        index = shelve.open(self.indexName(fieldname))
        try:
            keyval  = newdata[fieldname]  # в записях есть все поля
```

```
            reclist = index[keyval]       # получить, модифицировать, переписать
            reclist.append(recfilename)   # добавить в текущий список
            index[keyval] = reclist
        except KeyError:
            index[keyval] = [recfilename] # добавить в новый список

    def safeLoadKeysList(self, fieldname):
        if fieldname in self.indexfields:
            keys = shelve.open(self.indexName(fieldname)).keys()
            keys.sort()
        else:
            keys, index = self.loadIndexedTable(fieldname)
        return keys

    def safeLoadByKey(self, fieldname, fieldvalue):
        if fieldname in self.indexfields:
            dbase = shelve.open(self.indexName(fieldname))
            try:
                index = dbase[fieldvalue]
                reports = []
                for filename in index:
                    pathname = self.dirname + filename + '.data'
                    reports.append(pickle.load(open(pathname, 'r')))
                return reports
            except KeyError:
                return []
        else:
            key, index = self.loadIndexedTable(fieldname)
            try:
                return index[fieldvalue]
            except KeyError:
                return []

    # интерфейсы верхнего уровня (плюс dbcommon и dbfiles)

    def writeItem(self, newdata):
        # расширить для обновления индексов
        filename = self.cacheKey()
        dbfiles.Dbase.writeItem(self, newdata)
        for fieldname in self.indexfields:
            self.exclusiveAction(self.safeWriteIndex,
                                 fieldname, newdata, filename)

    def loadKeysList(self, fieldname):
        # загрузить только список ключей полей
        return self.sharedAction(self.safeLoadKeysList, fieldname)

    def loadByKey(self, fieldname, fieldvalue):
        # загрузить только список соответствующих записей
        return self.sharedAction(self.safeLoadByKey, fieldname, fieldvalue)

class DbaseErrata(Dbase):
    dirname     = 'DbaseFilesIndexed/errataDB/'
    filename    = dirname + 'indexes'
    indexfields = ['Submitter name', 'Submit date', 'Report state']

class DbaseComment(Dbase):
    dirname     = 'DbaseFilesIndexed/commentDB/'
    filename    = dirname + 'indexes'
    indexfields = ['Submitter name', 'Report state']    # индексировать по этим
```

```
#
# самотестирование
#

if __name__ == '__main__':
    import os
    dbase = DbaseComment()
    os.system('rm %s*'        % dbase.dirname)          # пустой каталог dbase
    os.system('echo > %s.lck' % dbase.filename)         # инициализировать файл блокировки
    # 3 записи; обычно есть email отправителя и описание, без страницы
    # дату передачи и состояние сообщения добавляет автоматически
    # метод сохранения записи
    records = [{'Submitter name': 'Bob',   'Page': 38, 'Submit mode': ''},
               {'Submitter name': 'Brian', 'Page': 40, 'Submit mode': ''},
               {'Submitter name': 'Bob',   'Page': 42, 'Submit mode': 'email'}]
    for rec in records: dbase.storeItem(rec)

    dashes = '-'*80
    def one(item):
        print dashes; print item
    def all(list):
        print dashes
        for x in list: print x

    one('old stuff')
    all(dbase.loadSortedTable('Submitter name'))          # загрузить плоский список
    all(dbase.loadIndexedTable('Submitter name'))         # загрузка сгруппированных
   #one(dbase.loadIndexedTable('Submitter name')[0])
   #all(dbase.loadIndexedTable('Submitter name')[1]['Bob'])
   #all(dbase.loadIndexedTable('Submitter name')[1]['Brian'])

    one('new stuff')
    one(dbase.loadKeysList('Submitter name'))             # bob, brian
    all(dbase.loadByKey('Submitter name', 'Bob'))         # соответствуют две записи
    all(dbase.loadByKey('Submitter name', 'Brian'))       # соответствует одна запись
    one(dbase.loadKeysList('Report state'))               # соответствуют все записи
    all(dbase.loadByKey('Report state',   'Not yet verified'))

    one('boundary cases')
    all(dbase.loadByKey('Submit mode', ''))               # не индексированная: загрузка
    one(dbase.loadByKey('Report state', 'Nonesuch'))      # неизвестное значение: []
    try:    dbase.loadByKey('Nonesuch', 'Nonesuch')       # плохие поля: исключение
    except: print 'Nonesuch failed'
```

Код этого модуля представляет собой нечто вроде исполняемого прототипа, но это здесь, собственно, и нужно. То, что фактически можно проводить эксперименты с кодом Python, способствует раннему обнаружению проблем с моделью.

Например, мне пришлось здесь переопределить метод `makeKey`, чтобы помещать имена файлов в локальный буфер (они также нужны для индексных полок). Это не совсем правильно, и если бы я собрался принять такой интерфейс базы данных, я, вероятно, изменил бы класс файла, чтобы он возвращал сгенерированные имена файлов, а не отбрасывал их. Такие несоответствия часто можно обнаружить только при написании настоящего кода – задача, оптимизируемая Python по замыслу.

Если запустить этот модуль в качестве сценария верхнего уровня, выполняется код самотестирования в конце файла, вывод которого приводится ниже. Здесь нет места для подробного ее объяснения, поэтому попытайтесь сопоставить ее с кодом самотестирования и проследить, как обрабатываются запросы с индексами полей и без них:

```
[mark@toy .../Internet/Cgi-Web/PyErrata/AdminTools]$ python dbaseindexed.py
--------------------------------------------------------------------------------
old stuff
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Rep
ort state': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Brian'}
--------------------------------------------------------------------------------
['Bob', 'Brian']
{'Bob': [{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '',
'Report state': 'Not yet verified', 'Submitter name': 'Bob'}, {'Submit date': '2
000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Report state': 'Not y
et verified', 'Submitter name': 'Bob'}], 'Brian': [{'Submit date': '2000/06/13,
11:45:01', 'Page': 40, 'Submit mode': '', 'Report state': 'Not yet verified', 'S
ubmitter name': 'Brian'}]}
--------------------------------------------------------------------------------
new stuff
--------------------------------------------------------------------------------
['Bob', 'Brian']
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Rep
ort state': 'Not yet verified', 'Submitter name': 'Bob'}
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Brian'}
--------------------------------------------------------------------------------
['Not yet verified']
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Brian'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Rep
ort state': 'Not yet verified', 'Submitter name': 'Bob'}
--------------------------------------------------------------------------------
boundary cases
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Brian'}
--------------------------------------------------------------------------------
[]
Nonesuch failed

[mark@toy .../PyErrata/AdminTools]$ ls DbaseFilesIndexed/commentDB/
960918301.263-895.data  960918301.506-895.data  Submitter-name  indexes.log
960918301.42-895.data   Report-state            indexes.lck

[mark@toy .../PyErrata/AdminTools]$ more DbaseFilesIndexed/commentDB/indexes.log
960918301.266   Requested: 895, writer
960918301.266   Aquired: 895
960918301.36    Released: 895
```

```
960918301.36    Requested: 895, writer
960918301.361   Aquired: 895
960918301.419   Released: 895
960918301.422   Requested: 895, writer
960918301.422   Aquired: 895
960918301.46    Released: 895
...еще строки...
```

Одним из недостатков этого интерфейса является то, что он работает только на машине, поддерживающей вызов `fcntl.flock` (обратите внимание, что предыдущий тест выполнялся под Linux). При желании использовать эти классы для поддержки индексированных баз данных файлов/полок на других машинах можно удалить этот вызов или сделать для него заглушку в модуле `mutex`, чтобы он возвращался, ничего не делая. При этом не получится надежных обновлений, но во многих приложениях это неважно:

```
try:
    import fcntl
    from FCNTL import *
except ImportError:
    class fakeFcntl:
        def flock(self, fileno, flag): return
    fcntl = fakeFcntl()
    LOCK_SH = LOCK_EX = LOCK_UN = 0
```

Вместо этого можно организовать `MutexCntl.lockFile` так, чтобы он ничего не делал при наличии флага, передаваемого аргументом командной строки, смешать с другим классом `MutexCntl` в конце, который ничего не делает при вызовах блокировок, или поискать механизмы блокировки, специфические для платформы (например, пакет расширений Windows экспортирует вызов блокировки файлов только для Windows; подробности в документации).

Независимо от того, используется ли блокировка, схема `dbaseindexed` плоских файлов вместе с индексированием по нескольким полкам может ускорить доступ по ключам для больших баз данных. Однако она требует также изменения логики сценария CGI верхнего уровня, реализующей вывод «Index only», и потому действует не так гладко. Она может также плохо работать с очень большими базами данных, когда информация записей распределяется по нескольким файлам. При необходимости можно в конечном счете расширить классы баз данных, чтобы они взаимодействовали с настоящей СУБД, такой как Oracle, MySQL, PostGres или Gadfly (описано в главе 16).

Все эти варианты в какой-то мере являются компромиссными, но тут мы опасно приблизились к тому, чтобы покинуть пределы этой главы. Так как при разработке модулей баз данных PyErrata не ставилась задача обеспечить их широкое применение или значительную масштабируемость, внесение дополнительных изменений предлагается в качестве упражнения.

15

Более сложные темы Интернета

«Путешествуя на плечах гигантов»

Эта глава завершает наше рассмотрение программирования на Python для Интернета изучением нескольких связанных с Интернетом тем и пакетов. В предыдущих пяти главах мы охватили многие вопросы – основы сокетов, написание сценариев для клиентов и серверов и программирование развитых сайтов с помощью Python. Однако мы еще не ознакомились с работой многих стандартных встроенных модулей Python. Кроме того, существует богатое собрание расширений сторонних разработчиков для создания веб-сценариев с помощью Python, которого мы вообще не касались.

В этой главе мы исследуем уйму дополнительных инструментов и расширений сторонних разработчиков, которые представляют интерес для интернет-программистов на Python. Попутно мы познакомимся с большими интернет-пакетами, такими как HTMLgen, JPython, Zope, PSP, Active Scripting и Grail. Мы изучим также стандартные инструменты Python, полезные интернет-программистам, в том числе режим Python ограниченного выполнения, поддержку XML, интерфейсы COM и технологии реализации собственных серверов. Помимо своей практической пользы эти системы демонстрируют, какие возможности открывает соединение веб с таким мощным объектно-ориентированным языком сценариев, как Python.

Перед началом сделаем объявление: все обсуждаемые темы представлены без лишних подробностей, а о некоторых интересных Интернет-системах наверняка вообще ничего не будет сказано. Кроме того, Интернет развивается с быстротою молнии, поэтому после выхода этой книги из печати, несомненно, появятся новые инструменты и технологии. В действительности большинство описываемых в этой главе систем появилось в течение пяти лет после первого издания этой книги, и следующие пять лет обещают оказаться столь же плодотворными. Как всегда, действует стандартное предупреждение о движущейся цели: читайте о пропущенных нами деталях в интернет-разделе библиотечного руководства Python и держите контакт с сообществом Python на *http://www.python.org*, чтобы иметь информацию о расширениях, которые здесь не освещаются за недостатком места или проницательности.

Zope: среда для создания публикаций в веб

Zope является сервером веб-приложений и инструментальным набором с открытым исходным кодом (open source), написанным на Python и допускающим индивидуальную настройку. Это технология для сервера, позволяющая веб-дизайнерам реализовывать сайты и приложения путем размещения иерархий объектов Python в Сети. С помощью Zope программисты могут сосредоточиться на создании объектов и возложить на Zope обработку базовых деталей HTTP и CGI. Если вас интересует создание более сложных сайтов, чем диалоги на основе форм, рассматривавшиеся нами в по-

следних трех главах, вам следует изучить Zope: с его помощью можно избавиться от решения многих задач, с которыми постоянно приходится сталкиваться программистам сценариев для веб.

Zope иногда сравнивают с коммерческими инструментальными пакетами для веб, такими как ColdFusion, при этом он бесплатно распространяется через Интернет компанией под названием Digital Creations и собрал вокруг себя большое и очень активное сообщество разработчиков. Действительно, многих участников недавней конференции по Python привлек Zope, явившийся отдельной темой конференции. Zope так широко распространился, что многие «питонисты» рассматривают его как программу-«приманку» («killer application») – систему, настолько удачную, что она естественным образом выдвигает Python в поле зрения разработчика. По крайней мере, Zope предлагает новый способ разработки сайтов для Сети значительно более высокого уровня, чем простые сценарии CGI.[1]

Компоненты Zope

Zope начал свое существование как набор инструментов (часть которого называлась «Bobo»), который Digital Creations разместила как публичную собственность. С тех пор он превратился в большую систему со многими составляющими, с растущим числом расширений (add-ons), называемых «продуктами» на жаргоне Zope, и довольно крутой кривой обучения. Мы не можем уделить ему достойное место в этой книге, но так как на момент ее написания Zope является одним из наиболее популярных приложений, основанных на Python, я сделал бы упущение, не сообщив некоторые касающиеся его детали.

Если говорить о базовых компонентах, то Zope состоит из следующих частей:

Брокер запросов к объектам (ORB) Zope

Центральной частью Zope является ORB, который отправляет поступившие запросы HTTP объектам Python и возвращает результаты автору запроса, действуя как постоянно выполняющийся посредник между областью HTTP CGI и вашим объектами Python. ORB пакета Zope подробнее описывается в следующем разделе.

Шаблоны документов HTML

Zope дает простой способ определить веб-страницы в качестве шаблонов, в которые автоматически вставляются значения из объектов Python. Шаблоны позволяют определить представление объекта в HTML независимо от реализации объекта. Например, значения атрибутов в объекте экземпляра класса могут автоматически включаться в текст шаблона по имени. Программисты шаблонов необязательно должны быть программистами Python, и наоборот.

Объектная база данных

Чтобы хранить данные постоянно, в Zope есть полная объектно-ориентированная система базы данных для хранения объектов Python. Объектная база данных Zope основана на модуле сериализации Python `pickle`, с которым мы познакомимся в следующей части книги, но кроме того, поддерживаются транзакции, отложенная загрузка объектов (lazy object fetches), которую иногда называют отсроченным вычислением (delayed evaluation), одновременный доступ и др. Объекты сохраня-

[1] На протяжении ряда лет обозреватели указывали и на другие системы как возможные приманки для Python, в том числе Grail, поддержку Python COM в Windows и JPython. Вероятно, они правы, и я вполне допускаю появление новых приманок после выхода этого издания. Но в тот момент, когда пишется эта книга, Zope вызывает ошеломляющий интерес как у разработчиков, так и у инвесторов.

ются и извлекаются по ключу, подобно тому как это делается с помощью стандартного модуля Python `shelve`, но классы должны быть подклассами импортируемого суперкласса `Persistent`, а хранилища объектов являются экземплярами импортируемого объекта `PickleDictionary`. Zope открывает и фиксирует транзакции в начале и конце запросов HTTP.

В состав Zope входит также управляющая среда для администрирования сайтов и API продуктов для упаковки компонентов. Zope поставляется вместе с этими и другими компонентами, интегрированными в целую систему, но каждая часть может также использоваться самостоятельно. Например, объектная база данных Zope может отдельно использоваться в произвольных приложениях Python.

Что такое публикация объектов?

Возможно, вам, как и мне, понятие публикации объектов в веб на первый взгляд может показаться несколько неясным, но в Zope это довольно просто: ORB Zope автоматически отображает URL, которые запрашивает HTTP, в вызовы объектов Python. Посмотрите на модуль Python и функцию в примере 15.1.

Пример 15.1. PP2E\Internet\Other\messages.py

```
"Модуль Python, опубликованный в веб с помощью Zope"

def greeting(size='brief', topic='zope'):
    "a published Python function"
    return 'A %s %s introduction' % (size, topic)
```

Это обычный код Python без каких-либо ссылок на Zope, CGI или Интернет в целом. Определенную в нем функцию можно, как обычно, вызвать из интерактивного приглашения:

```
C:\...\PP2E\Internet\Other>python
>>> import messages
>>> messages.greeting()
'A brief zope introduction'

>>> messages.greeting(size='short')
'A short zope introduction'

>>> messages.greeting(size='tiny', topic='ORB')
'A tiny ORB introduction'
```

Но если поместить этот файл модуля вместе со вспомогательными файлами Zope в надлежащий каталог на сервере, где выполняется Zope, он автоматически становится видимым в веб. Это значит, что функция становится опубликованным объектом – ее можно вызывать через URL, а возвращаемое ей значение становится страницей ответа. Например, если этот файл помещен в каталог `cgi-bin` на сервере `myserver.net`, то следующие URL эквивалентны трем приведенным выше вызовам:

```
http://www.myserver.net/cgi-bin/messages/greeting
http://www.myserver.net/cgi-bin/messages/greeting?size=short
http://www.myserver.net/cgi-bin/messages/greeting?size=tiny&topic=ORB
```

Когда к нашей функции осуществляется доступ по URL через веб таким способом, Zope ORB выполняет два волшебных трюка:

- URL автоматически транслируется в вызов функции Python. Первая часть URL после пути каталога (messages) указывает модуль Python, вторая (greeting) указывает функцию или другой вызываемый объект в модуле, а параметры после `?` становятся аргументами ключевых слов, передаваемых заданной функции.

- После выполнения функции возвращаемое ей значение автоматически появляется на новой странице в веб-броузере. Zope проделывает всю работу по форматированию результата как действительного ответа HTTP.

Иными словами, URL в Zope становится *удаленным вызовом функции*, а не просто вызовом сценария. Функции (и методы), вызываемые путем обращения к URL, кодируются на Python и могут находиться в произвольном месте Сети. Это как если бы сам Интернет стал пространством имен Python с одним каталогом модулей для каждого сервера.

Zope представляет собой серверную технологию, основанную на объектах, а не на текстовых потоках. Главное преимущество этой схемы в том, что деталями ввода и вывода CGI занимается Zope, в то время как программисты могут сосредоточиться на написании объектов области приложения, а не на генерации текста. При доступе к нашей функции через URL Zope автоматически находит указанный объект, переводит входные параметры в аргументы вызова функции, выполняет функцию и на основе возвращенного ей значения генерирует ответ HTTP. В целом, URL вида:

```
http://servername/dirpath/module/object1/object2/method?arg1=val1&arg2=val2
```

отображается Zope ORB, выполняющимся на `servername`, в вызов объекта Python в файле модуля Python, установленном в `dirpath`, принимая вид:

```
module.object1.object2.method(arg1=val1, arg2=val2)
```

Возвращаемое значение форматируется в страницу ответа HTML, возвращаемую клиенту, пославшему запрос (обычно броузеру). При использовании более длинных путей программы могут опубликовывать целые иерархии объектов; Zope просто использует общие протоколы Python доступа к объектам для получения объектов, расположенных вдоль пути.

Как обычно, URL, вроде перечисленных здесь, могут появляться в виде текста гиперссылки, вводимого в броузер вручную или используемого в запросе HTTP, генерируемом программой (например, с помощью модуля Python `urllib` в сценарии клиента). Параметры перечисляются непосредственно в конце этих URL, но если отправить данные по этому URL с помощью формы, будет достигнут тот же результат:

```
<form action="http://www.myserver.net/cgi-bin/messages/greeting" method=POST>
    Size:  <input type=text name=size>
    Topic: <input type=text name=topic value=zope>
    <input type=submit>
</form>
```

Здесь атрибут `action` вновь ссылается на URL нашей функции; когда пользователь заполнит эту форму и нажмет кнопку submit, входные данные формы, отправленные броузером, снова появятся в виде аргументов функции. Эти данные введены пользователем, а не закодированы жестко в конце URL, но нашей опубликованной функции не приходится об этом беспокоиться. В действительности Zope распознает несколько источников параметров и всегда транслирует их в аргументы функций или методов Python: входные данные формы, параметры в конце URL, заголовки HTTP и cookies, переменные окружения CGI и другие.

Однако это лишь малая часть того, что могут делать опубликованные объекты. Например, опубликованные функции и методы с помощью объектной базы данных Zope могут сохранять состояние, а кроме того, Zope предоставляет многие другие развитые средства, такие как поддержка отладки, готовые серверы HTTP, используемые вместе с ORB, и более детальное управление ответами на запросы по URL.

Все, что касается Zope, можно найти на *http://www.zope.org*. Там есть свежие версии программного обеспечения, а также разнообразная документация – от учебников до справочников и развитых примеров сайтов Zope. Кроме того, на прилагаемом к книге CD имеется дистрибутив Zope в той версии, которая имелась на момент сдачи книги в печать.

Создатель Python Гвидо ван Россум и его команда Pythonlabs главных разработчиков Python перешли из BeOpen в Digital Creations, родину среды Zope, с которой мы здесь знакомимся. Хотя сам Python остается системой open source, присутствие Guido в Digital Creations видится как стратегический ход, который поощрит дальнейший рост как Zope, так и Python.

HTMLgen: веб-страницы, создаваемые объектами

Одной из причин сложности сценариев CGI является их внутренняя зависимость от HTML: они должны встраивать и генерировать действительный код HTML для построения интерфейсов пользователя. Эти задачи решались бы проще, если бы синтаксис HTML был бы каким-то образом удален из сценариев CGI и обрабатывался с помощью внешнего инструмента.

HTMLgen является инструментом Python стороннего разработчика, созданным для удовлетворения этой потребности. С его помощью программы строят веб-страницы, создавая деревья объектов Python, которые представляют нужные страницы и «умеют» форматировать себя в виде HTML. Когда дерево создано, программа предлагает вершине дерева объектов Python сгенерировать для себя HTML, и на свет появляется законченная, правильно сформатированная веб-страница HTML.

Программам, генерирующим страницы с помощью HTMLgen, не требуется иметь дело с синтаксисом HTML; вместо этого они могут воспользоваться моделью объектов более высокого уровня, предоставляемой HTMLgen, и доверить ей этап форматирования. HTMLgen может использоваться в любом контексте, где требуется генерировать HTML. Особенно подходит HTMLgen для периодической генерации HTML из статических данных, но можно и создавать с его помощью HTML в сценариях CGI (хотя использование в контексте CGI влечет некоторую потерю в скорости). Например, применение HTMLgen идеально для задачи ежевечерней генерации веб-страниц по содержимому базы данных. HTMLgen можно также использовать для генерации документов, которых вообще нет в веб; создаваемый код HTML так же хорошо работает в автономном режиме.

Краткий учебник по HTMLgen

Мы не имеем здесь возможности глубоко изучать HTMLgen, но рассмотрим несколько простых примеров, чтобы составить себе представление о системе. HTMLgen поставляется в виде совокупности модулей Python, которые должны быть установлены на машине. После установки нужно просто импортировать из модуля `HTMLgen` объекты, соответствующие тегу, который должен быть сгенерирован, и создать экземпляры:

```
C:\Stuff\HTMLgen\HTMLgen>python
>>> from HTMLgen import *
>>> p = Paragraph("Making pages from objects is easy\n")
>>> p
<HTMLgen.Paragraph instance at 7dbb00>
>>> print p
```

```
<P>Making pages from objects is easy
</P>
```

Здесь мы создаем объект `HTMLgen.Paragraph` (экземпляр класса), передавая ему текст, который должен быть отформатирован. Все объекты HTMLgen реализуют методы `__str__` и могут выдавать для себя правильный код HTML. При печати объекта `Paragraph` выводится конструкция параграфа HTML. Объекты HTMLgen определяют также методы `append`, которые правильно действуют в соответствии с типом объекта; `Paragraph` просто помещает добавляемый текст в конец блока текста:

```
>>> p.append("Special < characters > are & escaped")
>>> print p
<P>Making pages from objects is easy
Special &lt; characters &gt; are &amp; escaped</P>
```

Обратите внимание, что HTMLgen преобразовал специальные символы (например, `<` означает `<`) для получения правильного HTML – не нужно беспокоиться о том, чтобы самому писать HTML или escape-коды. В HTMLgen есть классы для всех тегов HTML; вот пример работы объекта `List`, создающего упорядоченный список:

```
>>> choices = ['python', 'tcl', 'perl']
>>> print List(choices)
<UL>
<LI>python
<LI>tcl
<LI>perl
</UL>
```

В целом, HTMLgen умеет правильно интерпретировать передаваемые ему структуры данных. Например, встроенные последовательности автоматически отображаются в код HTML, который выводит вложенные списки:

```
>>> choices = ['tools', ['python', 'c++'], 'food', ['spam', 'eggs']]
>>> l = List(choices)
>>> print l
<UL>
<LI>tools
    <UL>
    <LI>python
    <LI>c++
    </UL>
<LI>food
    <UL>
    <LI>spam
    <LI>eggs
    </UL>
</UL>
```

Гиперссылки создаются тоже просто: создайте и выведите объект `Href` с адресом ссылки и текстом. (Аргумент с текстом можно заменить графическим изображением, как будет показано в примере 15.3.)

```
>>> h = Href('http://www.python.org', 'python')
>>> print h
<A HREF="http://www.python.org">python</A>
```

Чтобы сгенерировать HTML для целой страницы, создается один из объектов документов HTML, к нему добавляются составляющие его объекты и объект выводится. HTMLgen выдает полный код страницы, готовый к просмотру в броузере:

```
>>> d = SimpleDocument(title='My doc')
>>> p = Paragraph('Web pages made easy')
>>> d.append(p)
>>> d.append(h)
>>> print d
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<!-- This file generated using Python HTMLgen module. -->
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen 2.2.2">
        <TITLE>My doc</TITLE>
</HEAD>
<BODY>
<P>Web pages made easy</P>

<A HREF="http://www.python.org">python</A>

</BODY> </HTML>
```

Есть и другие классы документов, в том числе `SeriesDocument`, реализующий стандартную структуру в последовательности страниц. `SimpleDocument` действительно прост: он, в сущности, служит контейнером для других компонентов и генерирует надлежащую оболочку кода HTML. HTMLgen также предоставляет такие классы, как `Table`, `Form` и т. д., по одному для каждого типа конструкций HTML.

Обычно HTMLgen используется в сценарии, и генерируемый HTML помещается в файл или пересылается через интернет-соединение в контексте приложения CGI (вспомните, что выводимый текст в среде сценария CGI поступает в броузер). Сценарий примера 15.2 делает примерно то же, что мы только что выполняли интерактивно, но сохраняет выведенный текст в файле.

Пример 15.2. PP2E\Internet\Other\htmlgen101.py

```
import sys
from HTMLgen import *

p = Paragraph('Making pages from objects is easy.\n')
p.append('Special < characters > are & escaped')

choices = ['tools', ['python', 'c++'], 'food', ['spam', 'eggs']]
l = List(choices)

s = SimpleDocument(title="HTMLgen 101")
s.append(Heading(1, 'Basic tags'))
s.append(p)
s.append(l)
s.append(HR())
s.append(Href('http://www.python.org', 'Python home page'))

if len(sys.argv) == 1:
    print s                 # послать html в sys.stdout или файл
else:
    open(sys.argv[1], 'w').write(str(s))
```

В этом сценарии также используются объекты `HR` для образования горизонтальной линии и `Heading` для вставки строки заголовка. Он выводит HTML в стандартный выходной поток (если не задано аргументов) или в явно указанный файл; встроенная функция `str` вызывает методы объекта `__str__`, так же как и `print`. Запустить этот сценарий из командной строки системы, чтобы создать файл, можно одним из способов:

```
C:\...\PP2E\Internet\Other>python htmlgen101.py > htmlgen101.html
C:\...\PP2E\Internet\Other>python htmlgen101.py htmlgen101.html
```

В обоих случаях вывод сценария представляет собой файл с законной страницей HTML, которую можно просмотреть в броузере, введя имя файла в поле адреса или щелкнув по нему в менеджере файлов. Он должен выглядеть, примерно как на рис. 15.1.

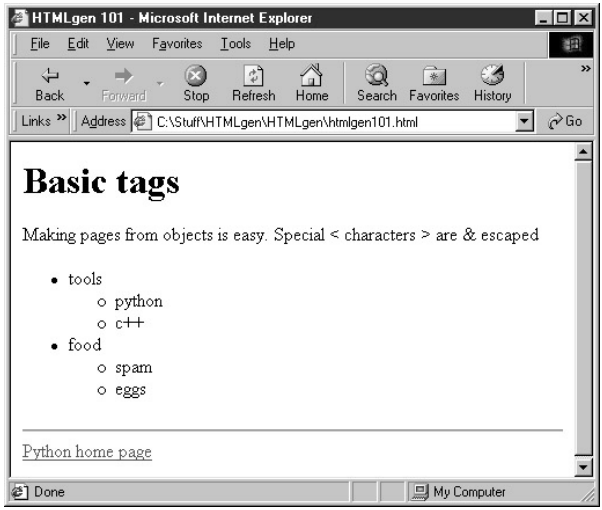


Рис. 15.1. Просмотр в броузере вывода htmlgen101.py

Посмотрите на файл *htmlgen101.html* в дистрибутиве примеров, если хотите непосредственно увидеть HTML, сгенерированный для описания этой страницы (он весьма похож на вывод предыдущего документа). В примере 15.3 показан другой сценарий, выполняющий нечто менее жестко определенное: он строит веб-страницу, показывающую его собственный код.

Пример 15.3. PP2E\Internet\Other\htmlgen101-b.py

```
import sys
from HTMLgen import *
d = SimpleDocument(title="HTMLgen 101 B")

# показать этот сценарий
text = open('htmlgen101-b.py', 'r').read()
d.append(Heading(1, 'Source code'))
d.append(Paragraph( PRE(text) ))

# добавить gif и ссылки
site  = 'http://www.python.org'
gif   = 'PythonPoweredSmall.gif'
image = Image(gif, alt='picture', align='left', hspace=10, border=0)

d.append(HR())
d.append(Href(site, image))
d.append(Href(site, 'Python home page'))

if len(sys.argv) == 1:
    print d
else:
    open(sys.argv[1], 'w').write(str(d))
```

Здесь объект PRE задает преформатированный текст, а объект Image генерирует код, выводящий файл GIF на создаваемой странице. Обратите внимание, что параметры тега HTML, такие как alt и align, при создании объектов HTMLgen указываются как аргументы ключевых слов. Выполнив этот сценарий и открыв созданный им вывод в броузере, получим страницу, показанную на рис. 15.2; графическое изображение внизу тоже служит гиперссылкой, потому что было помещено внутрь объекта Href.

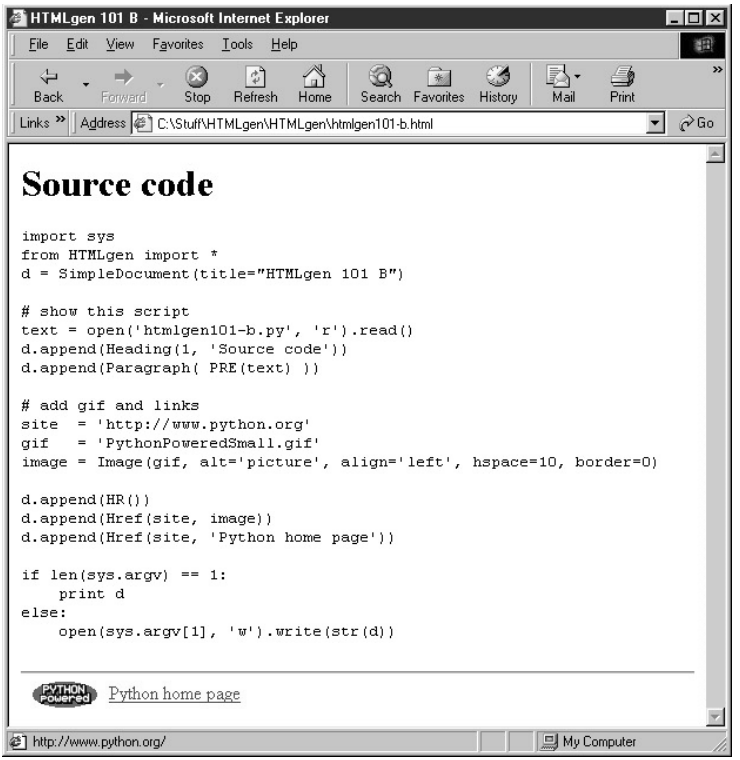


Рис. 15.2. Просмотр вывода htmlgen101-b.py в броузере

Вот и все (плюс несколько хороших развитых функций), что касается использования HTMLgen. После более близкого знакомства с ним можно создавать веб-страницы с помощью кода Python, не вводя вручную тегов HTML. Конечно, HTMLgen требует писать код, а не перетаскивать объекты на страницу мышкой, но этот код чрезвычайно прост и при необходимости динамического создания страниц позволяет использовать более сложную программную логику.

На самом деле, теперь, познакомившись с HTMLgen, вы увидите, что многие из файлов HTML, показанных ранее в книге, можно упростить, если использовать в них HTMLgen, а не непосредственно код HTML. Предшествовавшие сценарии CGI также могли бы использовать HTMLgen, хотя и с потерей скорости – непосредственный вывод текста происходит быстрее, чем при генерации его из деревьев объектов, хотя, возможно, незначительно (сценарии CGI обычно ограничены скоростью передачи данных в сети, а не скоростью CPU).

HTMLgen является программным обеспечением open source, но не входит стандартно в Python, а потому требует отдельной установки. Экземпляр HTMLgen есть на прилагаемом к книге CD, но текущие местонахождение и версия его должны быть на сайте

Python. После установки добавьте путь к HTMLgen в значение своей переменной PYTHONPATH, чтобы получить доступ к его модулям. Дополнительную документацию по HTMLgen можно найти в самом пакете: в его подкаталоге *html* содержится руководство по HTMLgen в формате HTML.

JPython (Jython): Python для Java

JPython (переименован в «Jython») является совершенно особой реализацией языка программирования Python, позволяющей программистам использовать Python как составную часть для написания сценариев в приложениях, основанных на Java. Коротко говоря, JPython делает код Python похожим на Java, в результате чего появляется ряд технологических возможностей, унаследованных из области Java. С помощью JPython код Python может выполняться как клиентские апплеты в веб-броузерах, как сценарии сервера и в ряде других качеств. JPython отличается от других упоминавшихся в этом разделе систем по области своего действия: будучи основан на базовом языке Python, с которым мы имеем дело в этой книге, он в действительности не расширяет его, а заменяет базовую реализацию.[1]

В этом разделе кратко рассматривается JPython и отмечаются некоторые основания, по которым его применение вместо стандартной реализации Python может быть желательным или нежелательным. Хотя JPython интересен главным образом программистам, пишущим приложения на Java, он подчеркивает возможности интеграции и проблемы определения языка, которые заслуживают внимания всех пользователей Python. Поскольку JPython сосредоточен вокруг Java, чтобы хорошо понять JPython, требуются некоторые знания о разработке на Java, и эта книга не претендует на то, чтобы обучить ему на протяжении нескольких следующих страниц. Заинтересованным читателям за дополнительными сведениями следует обратиться к другим материалам, в том числе документации JPython на *http://www.jpython.org*.

Реализация JPython сейчас называется «Jython». Хотя вы, вероятно, по-прежнему некоторое время будете видеть его в Сети (и этой книге) под первоначальным именем JPython, новое название Jython со временем должно распространиться.

Краткое знакомство с JPython

Функционально JPython представляет собой совокупность классов Java, выполняющих код Python. Он состоит из написанного на Java компилятора Python, транслирующего сценарии Python в байт-код Java, которые могут выполняться *виртуальной машиной Java* – компонентом этапа исполнения, выполняющим программы Java и имеющимся в основных веб-броузерах. Кроме того, JPython автоматически предоставляет все библиотеки классов Java для использования в сценариях Python. Вот краткий состав системы JPython:

[1] На момент написания книги JPython является второй реализацией языка Python. По контрасту стандартную первоначальную реализацию Python сейчас иногда называют «CPython», поскольку она реализована на ANSI C. Среди прочего, реализация JPython стимулирует более ясное определение самого языка Python, независимое от действия конкретных реализаций. Готовится также новая реализация Python для среды Microsoft C#/.NET (см. ниже в этой главе), которая может оказать дальнейшее влияние на определение того, что же такое Python.

Компилятор Python в байт-код Java

JPython всегда компилирует исходный код Python в байт-код Java и передает его для исполнения механизму этапа исполнения виртуальной машины Java (JVM). Компилятор командной строки `jpythonc` тоже может транслировать исходный код Python в файлы Java *.class* и *.jar*, которые затем могут быть использованы как аплеты, компоненты JavaBeans, сервлеты и т. д.. Для JVM код Python, выполняемый через Jpython, выглядит так же, как код Java. Помимо создания возможности выполнения кода Python на JVM, код JPython наследует также все стороны системы этапа исполнения Java, в том числе уборку мусора Java и модели защиты. `jpythonc` налагает также правила классов исходных файлов Java.

Доступ к библиотекам классов Java (расширение)

JPython использует *отражение API* Java (информацию о типе в момент исполнения) для показа всех имеющихся библиотек классов Java сценариям Python. Это означает, что программы Python, написанные для системы JPython, могут автоматически обращаться к любому резидентному классу Java, просто импортируя его. Интерфейс компиляции Python в Java полностью автоматический и бесшовный – библиотеки классов Java выглядят так, как если бы были написаны на Python. Операция импорта в сценариях JPython могут относиться как к модулям Jpython, так и к библиотекам классов Java. Например, когда сценарий JPython импортирует `java.awt`, он получает доступ ко всем средствам, имеющимся в библиотеке `awt`. JPython внутренне создает «фиктивный» объект модуля Python, служащий интерфейсом к `awt` во время импорта. Этот фиктивный модуль состоит из хуков для отправки вызовов из кода JPython методам классов Java и автоматического преобразования типов данных между представлениями в Java и Python, если таковое потребуется. Для сценариев JPython библиотеки классов Java выглядят точно так же, как обычные модули Python (хотя их интерфейсы определены в области Java).

Объединенная объектная модель

Объекты JPython фактически внутренне являются объектами Java. В действительности JPython реализует типы Python как экземпляры класса Java `PyObject`. Напротив, в Python, реализованном на C, классы и типы в текущей версии различны. Например, в JPython число 123 является экземпляром класса Java `PyInteger`, и можно задавать такие вещи, как `[].__class__`, потому что все объекты являются экземплярами классов. Это упрощает преобразование данных между языками: Java может автоматически преобразовывать объекты Python, так как они являются объектами Java. JPython автоматически преобразует типы между языками в соответствии с отображением стандартных типов, если это требуется при обращении к библиотекам Java, и выбирает между сигнатурами перегружаемых методов Java.

API для запуска Python из Java (встраивание)

JPython предоставляет также интерфейсы, позволяющие программам Java выполнять код JPython. Как и при встраивании в C и C++, это позволяет выполнять индивидуальную настройку приложений Java с помощью участков динамически создаваемого кода JPython. Например, JPython поставляется с классом Java `PythonInterpreter`, позволяющим программам Java создавать объекты, действующие как пространства имен для выполняемого кода Python. Каждый объект `PythonInterpreter` можно грубо считать модулем Python с такими методами, как `exec` (выполнить строку кода Python), `execfile` (выполнить файл Python) и методами `get` и `set` для присвоения значений глобальным переменным Python. Так как объекты Python в действительности являются экземплярами класса Java `PyObject`, содержащий их слой Java может естественным образом обращаться к данным, создаваемым кодом Python, и обрабатывать их.

Интерактивная командная строка Python

Как и в стандартной реализации Python, в JPython есть интерактивная командная строка, выполняющая код сразу после его ввода. В Jpython программа `jpython` эквивалентна исполняемому модулю `python`, которым мы пользуемся в этой книге; при запуске без аргументов он открывает интерактивный сеанс. Помимо всего прочего, это позволяет программистам JPython импортировать и тестировать компоненты классов, фактически написанные на Java. Одной этой возможности достаточно, чтобы заинтересовать многих программистов Java.

Автоматизация интерфейсов

Библиотеки Java несколько проще использовать в коде Jpython, чем в Java. Это связано с тем, что JPython автоматизирует некоторые этапы кодирования, предполагаемые Java. Например, *обработчики обратного вызова* для библиотек Java GUI могут быть простыми функциями Python, хотя кодировщики Java должны предоставлять методы в полностью заданных классах (в Java нет «первоклассных» (first-class) объектов функций). JPython также делает объекты данных (*data members*) класса Java доступными как имена атрибутов Python (*`object.name`*) и аргументы ключевых слов конструктора объектов (*`name=value`*); такой синтаксис Python транслируется в вызовы методов доступа `getName` и `setName` в классах Java. Мы увидим эти приемы автоматизации в действии в последующих примерах. Вы не обязаны их использовать (и они могут смутить программистов Java с первого взгляда), но они еще больше упрощают программирование на JPython и придают библиотекам классов Java большее сходство с Python.

В результате всего этого JPython позволяет писать программы Python, которые могут выполняться на любой машине, поддерживающей Java, – в частности, в контексте большинства веб-броузеров. Еще важнее то, что поскольку программы Python транслируются в байт-код, JPython обеспечивает чрезвычайно гладкую и естественную интеграцию между этими двумя языками. Тот и другой действуют в рамках модели Java, поэтому вызовы, пересекающие языковые границы, осуществляются тривиально. Благодаря подходу, осуществляемому JPython, становится даже возможным создание подклассов для классов Java в Python и наоборот.

А зачем вообще все эти хлопоты с встраиванием Python в среду Java? Самый очевидный ответ заключается в том, что JPython облегчает использование компонентов Java: сценарии JPython обычно по размеру значительно меньше своих Java-эквивалентов и значительно менее сложны. В более широком смысле ответ такой же, как для среды C или C++: Python, как простой в использовании объектно-ориентированный язык сценариев, естественным образом дополняет язык программирования Java.

Сейчас большинству людей ясно, что Java слишком сложен, чтобы служить языком сценариев или быстрой разработки. Но именно в этом качестве преуспевает Python; смешение с Python, осуществляемое с помощью JPython, добавляет в Java-системы компонент сценариев, так же, как это происходит при интеграции Python с C или C++. Например, с помощью JPython можно быстро создавать прототипы систем Java, интерактивно тестировать классы Java и делать системы Java открытыми для настройки конечным пользователем. В целом введение Python в разработки на Java может значительно повысить производительность труда программистов, как это происходит с системами C и C++.

Простой пример использования JPython

Когда программа Python скомпилирована с помощью JPython, она превращается в чистый Java: программа транслируется в байт-код Java, использует при работе клас-

JPython в сравнении с Python C API

Функционально JPython является в основном интегрирующей системой: он позволяет смешивать Python с компонентами Java. В следующей части книги мы изучим способы интеграции Python с компонентами C и C++. Следует заметить, что для интеграции Python с Java требуются другие технологии (такие, как компилятор JPython), потому что Java довольно закрытая система: она предпочитает смесь, состоящую только из Java. Средства интеграции с C и C++ в целом менее ограничивающие в отношении допущений языка, и годятся любые компоненты C-совместимых языков. Строгость Java частично обусловлена задачами безопасности, но в итоге поощряются технологии интеграции, специфические только для Java.

С другой стороны, поскольку Java открывает информацию о типах в момент исполнения через отражение своего API, JPython может значительно автоматизировать преобразования и диспетчеризацию, необходимые для доступа к компонентам Java из сценариев Python; код Python просто импортирует и вызывает компоненты Java. При смешении Python с C или C++ необходимо создать слой «связующего» кода, который явно объединяет два языка. Отчасти это можно автоматизировать (с помощью системы SWIG, с которой мы познакомимся далее в этой книге). Никакого связующего кода в JPython, однако, не требуется, так как разработчики JPython (и Java) уже проделали всю работу по связыванию в общем виде. Существует также возможность смешивать компоненты C/C++ с Java через ее интерфейс машинно-зависимых вызовов (JNI), но это может оказаться обременительным и перечеркнуть хваленые преимущества переносимости и безопасности Java.

сы Java, и весь Python остается в первоначальном исходном коде. Поскольку сам компилятор написан на Java, JPython иногда называют «100-процентно чистым Java». Такой ярлык может более приличествовать продавцам, чем программистам, потому что сценарии JPython все равно пишутся с использованием стандартного синтаксиса Python. Скажем, пример 15.4 является законной программой JPython, основанной на примере, первоначально написанном Гвидо ван Россумом.

Пример 15.4. PP2E\Internet\Other\jpython.py

```
###############################################################################################
# Реализация простого калькулятора на JPython; вычисляется сразу все выражение с помощью
# встроенной функции Python eval() - компилятор Jpython присутствует во время выполнения
###############################################################################################

from java import awt                     # получить доступ к библиотекам классов Java
from pawt import swing                  # здесь они выглядят как модули Python

labels = ['0', '1', '2', '+',           # метки для кнопок калькулятора
          '3', '4', '5', '-',           # будут использоваться в сетке 4x4
          '6', '7', '8', '*',
          '9', '.', '=', '/' ]

keys = swing.JPanel(awt.GridLayout(4, 4))     # магия библиотеки классов Java
display = swing.JTextField()                  # автопреобразование данных Python в Java

def push(event):                              # обратный вызов для обычных клавиш
    display.replaceSelection(event.actionCommand)

def enter(event):                             # обратный вызов для клавиши '='
```

```
        display.text = str(eval(display.text))    # вычисление выражения с помощью eval() Python
        display.selectAll()

    for label in labels:                          # построить сетку из элементов кнопок
        key = swing.JButton(label)                # при нажатии вызов функций Python
        if label == '=':
            key.actionPerformed = enter
        else:
            key.actionPerformed = push
        keys.add(key)

    panel = swing.JPanel(awt.BorderLayout())      # создать панель swing
    panel.add("North", display)                   # текст и сетка клавиш в середине
    panel.add("Center", keys)
    swing.test(panel)                             # запустить в GUI
```

Во-первых, обратите внимание, что это настоящий код Python – в сценариях JPython используется тот же самый базовый язык, с которым мы работаем на протяжении этой книги. Это радует, потому что языком Python так просто пользоваться и потому, что не нужно учить новый язык сценариев, чтобы пользоваться JPython. Это также означает, что полностью доступен синтаксис Python как языка высокого уровня и его инструменты. Например, в этом сценарии встроенная функция Python `eval` используется для одновременного синтаксического анализа и вычисления создаваемого выражения, избавляя нас от необходимости писать процедуру вычисления выражений с чистого листа.

Приемы автоматизации интерфейсов

Предшествующий пример калькулятора иллюстрирует также два приема автоматизации интерфейсов, осуществляемые JPython: обратный вызов функций и преобразование атрибутов. Java-программисты, возможно, уже заметили, что в этом примере не используются классы. Подобно стандартному Python и в отличие от Java, JPython поддерживает ООП, но не обязывает к его использованию. Простые функции Python прекрасно действуют в качестве *обработчиков обратного вызова*. В примере 15.4 присвоение `key.actionPerformed` объекту функции Python оказывает такой же эффект, как регистрация экземпляра класса, определяющего метод обработчика обратного вызова:

```
def push(event):
    ...
key = swing.JButton(label)
key.actionPerformed = push
```

Это заметно проще, чем более близкое к стилю Java:

```
class handler(awt.event.ActionListener):
    def actionPerformed(self, event):
        ...
key = swing.JButton(label)
key.addActionListener(handler())
```

JPython автоматически отображает функции Python в модель обратных вызовов методов классов Java. Программисты Java, во-первых, могут недоумевать теперь, почему можно осуществлять присваивание чему-то с именем `key.actionPerformed`. Второй волшебный трюк Jpython состоит в том, чтобы заставить свойства классов Java выглядеть просто как *атрибуты объектов* в коде Python. В абстрактной форме код JPython вида:

```
X = Object(argument)
X.property = value + X.property
```

эквивалентен более традиционному и сложному стилю Java:

```
X = Object(argument)
X.setProperty(value + X.getProperty())
```

То есть JPython автоматически отображает присвоение атрибутов и ссылки в вызовы методов доступа Java, изучая сигнатуры классов Java (и, возможно, файлы Java BeanInfo, если они используются). Более того, свойствам можно присваивать значения с помощью *аргументов ключевых слов* в вызовах конструкторов объектов, поэтому:

```
X = Object(argument, property=value)
```

эквивалентно такой более традиционной форме:

```
X = Object(argument)
X.setProperty(value)
```

и следующей, зависящей от отображения имен атрибутов:

```
X = Object(argument)
X.property = value
```

Можно объединить автоматизацию обратного вызова и свойств, создав еще более простую версию фрагмента кода обратного вызова:

```
def push(event):
    ...
key = swing.JButton(label, actionPerformed=push)
```

Применять эти приемы автоматизации не обязательно, но, опять-таки, они делают сценарии JPython проще, а в этом главная цель соединения Python с Java.

Создание апплетов Java с помощью JPython

Я был бы не прав, не включив краткий пример кода Jpython, который прямо маскируется под *апплет* Java: код, который располагается на машине сервера, но загружается на машину клиента и выполняется на ней при обращении к его интернет-адресу. Стоящая за этим механика заключается в основном в создании подкласса надлежащего класса Java в сценарии JPython, как показано в примере 15.5.

Пример 15.5. PP2E\Internet\Other\jpython-applet.py

```
########################################
# a simple java applet coded in Python
########################################

from java.applet import Applet                      # получить суперкласс java

class Hello(Applet):
    def paint(self, gc):                            # обратный вызов paint
        gc.drawString("Hello applet world", 20, 30) # отобразить текст

if __name__ == '__main__':                          # при автономном выполнении
    import pawt                                     # получить библиотеку java awt
    pawt.test(Hello())                              # выполнять цикл awt
```

Класс Python в этом коде наследует весь необходимый протокол апплета из стандартного суперкласса Java `Applet`, поэтому здесь немного нового. В Jpython классы Python всегда могут образовывать подклассы классов Java, потому что объекты Python в действительности являются объектами Java после компиляции и запуска. Написанный на Python метод `paint` из этого сценария будет при необходимости автоматически

запущен из цикла событий Java AWT; он просто использует для вывода текстового сообщения передаваемый ему объект указателя интерфейса пользователя `gc`.

Если с помощью утилиты командной строки `jpythonc` откомпилировать это в файл Java *.class* и правильно сохранить этот файл на веб-сервере, можно использовать его в точности как апплеты, написанные на Java. Так как большинство веб-серверов содержит JVM, это означает, что такие сценарии Python можно использовать в качестве клиентских программ, создающих сложные средства пользовательских интерфейсов в броузере, и т. д.

Преимущества и недостатки JPython

Следует отметить одно не особо приятное для тех, кто не имеет соответствующей подготовки, свойство JPython: несмотря на то что описанные сценарии калькулятора и апплета написаны на «чистом» Python, использованные в них библиотеки совсем не те, с которыми мы встречались ранее. На самом деле применяемые библиотечные вызовы в корне различны. Например, калькулятор зависит в основном от импортированных библиотек классов Java, а не стандартных библиотек Python. Чтобы разобраться в этом коде, необходимо действительно понимать библиотеки Java `awt` и `swing`, и этот *сдвиг по библиотекам* между реализациями языка становится более острым по мере роста размера программ. Пример с апплетом еще более привязан к Java: он зависит как от библиотек Java интерфейса пользователя, так и от протоколов апплетов Java.

Если вы уже знакомы с библиотеками Java, то это, конечно, не проблема. Но так как большая часть работы, производимой реальными программами, осуществляется с помощью библиотек, факт зависимости большей части кода JPython от совершенно других библиотек делает совместимость со стандартным Python не столь убедительной, как может показаться на первый взгляд. Строго говоря, за исключением самых тривиальных примеров базового языка, программы JPython не будут выполняться в стандартном интерпретаторе Python, а многие стандартные программы Python не станут работать в JPython.

Обычно JPython предоставляет целый ряд компромиссных решений, отчасти из-за своей незрелости на момент написания этой книги. Я хочу заранее подчеркнуть, что JPython действительно является прекрасным средством написания сценариев для Java – можно утверждать, что лучшим из имеющихся, и большинство его компромиссов, вероятно, мало заботят разработчиков Java. Например, если вы пришли в JPython из Java, то центральное место библиотек Java в сценариях JPython может быть скорее достоинством, чем недостатком. Но если вам предоставляется выбор между стандартной и основанной на Java реализациями Python, следует знать о некоторых последствиях использования JPython:

JPython не полностью совместим со стандартным языком Python

> На момент написания этой книги JPython еще не совсем совместим со стандартным языком Python, как он определен в оригинальной реализации на C. В некоторых отдельных местах сам базовый язык Python действует иначе в JPython. Например, до самого недавнего времени присвоение объектов типа файла стандартному устройству ввода `sys.stdin` было невозможно, а исключительные ситуации оставались строками, а не объектами классов. Список несоответствий (который можно увидеть на *http://www.jpython.org*) со временем, вероятно, сократится, но никогда не исчезнет полностью. Более того, новые функции языка, скорее всего, появятся в Jpython позднее, чем в стандартной реализации, основанной на C.

JPython требует от программистов умения разработки на Java

Синтаксис языка составляет только один аспект программирования. Упомянутый выше сдвиг по библиотекам служит только одним примером зависимости Jpython от системы Java. Чтобы реально работать в JPython, вам придется не только изучить библиотеки Java, но и овладеть средой программирования Java в целом. На JPython перенесены многие стандартные библиотеки Python, и регулярно переносятся все новые. Но основные инструменты Python, такие как GUI Tkinter, возможно, никогда не появятся в JPython (их заменяют инструменты Java).[1] Кроме того, многие центральные свойства библиотек Python не могут поддерживаться в JPython, потому что они нарушили бы ограничения Java по безопасности. Например, вызов `os.system` для запуска команд оболочки не может появиться в JPython.

JPython применим только там, где установлена JVM

Для выполнения кода JPython требуется среда исполнения Java. С учетом распространенности Интернета может показаться, что это не проблема, но мне совсем недавно пришлось работать в нескольких компаниях, поставка которым приложений, выполняемых на JVM, оказалась невозможна. Попросту говоря, у клиента не оказалось JVM. В таких ситуациях нужно отказаться от использования Jpython или поставлять JVM вместе с приложением, чтобы выполнять откомпилированный код JPython. Поставка стандартной системы Python вместе со своими продуктами совершенно бесплатна; поставка JVM может потребовать лицензирования и платы. Это вызовет меньше забот, когда появится надежная виртуальная машина Java с открытым исходным кодом. Но если вы хотите использовать JPython сегодня и не уверены в том, что ваши клиенты смогут выполнять ваши системы в броузерах с поддержкой Java (или другими компонентами JVM), следует учесть возможную стоимость поставки среды исполнения Java вместе со своими продуктами.[2]

JPython не поддерживает модули расширения Python, написанные на C или C++

В настоящее время никакие модули расширения C или C++, написанные для C-реализации Python, не будут работать с JPython. Это существенная преграда для употребления JPython вне рамок приложений, выполняемых в броузере. К сегодняшнему дню сообществом пользователей Python, насчитывающим полмиллиона человек, разработаны тысячи расширений для Python на C, в значительной мере определяющих практическую ценность работы с Python. Альтернатива, которую предоставляет в данное время JPython, заключается в том, что выставляются библиотеки классов Java и программистам предлагается писать новые расширения на Java. Но при этом отбрасывается обширная библиотека прошлых и будущих наработок Python. В принципе, расширения на C можно поддерживать с помощью интерфейса платформозависимых вызовов Java (native call interface), но это сложно, не реализовано и может свести на нет переносимость и безопасность Java.

JPython существенно медленней, чем C Python

В настоящее время код Python обычно медленнее выполняется в реализации JPython. Насколько медленнее, зависит от теста, используемой для выполнения теста JVM, наличия JIT-компилятора и того, на кого вы ссылаетесь. В приводимых ре-

---

[1] Однако посмотрите на замечание, сделанное ниже, в конце раздела, посвященного Grail; начальный перенос Tkinter для JPython уже существует в Сети.

[2] Вам потребуется JVM и для разработки этих продуктов! Установка JPython под Windows 98 при написании этой книги потребовала усилий – не из-за JPython, а потому что пришлось столкнуться с командами Java, выполняемыми во время установки, а также найти и установить другую JVM вместо поставляемой Microsoft. В зависимости от платформы можно столкнуться с зависимостью JPython от Java еще до того, как вы введете первую строку кода.

зультатах испытаний указывается на замедление по сравнению с C Python от 1,7 раза до 10, и даже до 100 раз. Независимо от того, каково точное число, дополнительный слой логики, необходимый JPython для отображения Python в модель исполнения Java, влечет дополнительное уменьшение скорости для и без того не быстрой JVM и делает маловероятным достижение JPython такой же скорости, как у реализации C Python, когда-либо в будущем. С учетом того, что C Python медленнее, чем компилируемые языки типа C, дополнительное замедление JPython снижает его полезность вне области сценариев Java. Более того, библиотека GUI `Swing`, используемая сценариями JPython, обладает мощью, но считается самой медленной и громоздкой из всех Python GUI. Учитывая, что библиотека Python Tkinter является стандартным переносимым решением GUI, собственные средства Java для интерфейса пользователя сами по себе, вероятно, не оправдывают использование реализации JPython.

JPython менее устойчив, чем C Python

На момент написания этой книги JPython содержит заметно больше ошибок, чем стандартная C-реализация языка. Конечно, это объясняется молодостью языка и меньшим количеством пользователей и зависит от JVM, но в Jpython у вас больше шансов напороться на сук. Напротив, в C Python с появления его в 1990-м поразительно мало ошибок.

JPython менее переносим, чем C Python

Стоит также отметить, что на момент написания книги базовый язык Python значительно более переносим, чем Java (несмотря на противоположные утверждения рекламы). Поэтому развертывание стандартного кода Python с основанной на Java реализацией JPython может фактически *снизить* его переносимость. Естественно, это зависит от набора используемых расширений, но стандартный Python сегодня выполняется всюду – от карманных PDA до PC, суперкомпьютеров Cray и мэйнфреймов IBM.

Некоторые несовместимости между JPython и стандартным Python могут быть очень тонкими. Например, JPython целиком наследует поведение механизма исполнения Java, в том числе ограничения безопасности Java и уборку мусора. В Java уборка мусора не использует стандартную схему Python со счетчиком ссылок и потому может автоматически убирать рекурсивные объекты.[1] Это означает также, что некоторые распространенные идиомы программирования на Python не будут действовать. Вот, например, типичный код Python для обработки файлов в цикле:

```
for filename in bigfilenamelist:
    text = open(filename).read()
    dostuffwith(text)
```

Этот код работает, потому что файлы автоматически закрываются при уборке мусора в стандартном Python, и можно быть уверенным, что файловый объект, возвращенный вызовом `open`, немедленно попадет в уборку мусора (он временный, поэтому после вызова `read` ссылок на него не останется. Однако в JPython этот код работать не будет, потому что неизвестно, когда будет уничтожен временный объект файла. Чтобы не превысить допустимое число указателей файлов, обычно для Jpython этот код нужно написать по-другому:

```
for filename in bigfilenamelist:
    file = open(filename)
```

[1] Но в Python 2.0 сборщик мусора тоже может теперь убирать рекурсивные объекты. Смотрите примечания к версии 2.0 и приложение A «Последние изменения в Python».

```
text = file.read()
dostuffwith(text)
file.close()
```

С таким же несоответствием реализации можно столкнуться, если предположить, что выходные файлы сразу закрываются: `open(name,'w').write(bytes)` убирает и закрывает временный объект файла и потому сбрасывает байты из буфера в файл только в стандартной C-реализации Python, в то время как JPython убирает объект файла в произвольный момент времени в будущем. Кроме того, деструкторы классов Python `__del__` не вызываются в JPython из-за осложнений, связанных с завершением жизни объектов.

Выбирая Python

Из-за опасений, подобных перечисленным выше, реализация JPython языка Python лучше всего, вероятно, пригодна для использования только в контекстах, где интеграция с Java или взаимодействие с веб-броузером имеют решающее значение для проекта. Конечно, вам решать, но стандартная C-реализация представляется более подходящей для большинства остальных приложений Python. Тем не менее, для JPython остается весьма широкая область – почти все системы и программисты Java окажутся в выигрыше от добавления JPython в свой инструментарий.

JPython позволяет программистам писать программы, использующие библиотеки классов Java, значительно меньшие по объему и сложности, чем их эквиваленты на Java. Поэтому JPython успешно применяется в качестве языка расширения в системах, основанных на Java, особенно тех, которые будут выполняться в контексте веб-броузеров. Так как Java является стандартным компонентом большинства веб-броузеров, сценарии JPython часто выполняются автоматически без каких-либо действий по установке на машине клиента. Более того, даже приложения Java, не имеющие никакого отношения к веб, могут извлечь пользу из простоты использования JPython; гладкая интеграция с библиотеками классов Java превращает JPython в лучший из имеющихся сегодня инструментов для написания сценариев и тестирования Java.

Однако для большинства приложений стандартная реализация Python, возможно, интегрированная с компонентами C и C++, может оказаться лучшим проектным решением. Получаемая система, скорее всего, будет работать быстрее, меньше стоить в поставке, будет иметь доступ ко всем модулям расширения Python, окажется более устойчивой и переносимой и легче в сопровождении для тех, кто знаком со стандартным Python.

С другой стороны, я хочу снова подчеркнуть, что перечисленные альтернативы рассмотрены в основном с точки зрения Python, и для разработчика на Java, ищущего средство создания сценариев для систем, основанных на Java, многие из отмеченных недостатков могут быть малозначительны. И, честно говоря, некоторые из проблем Jpython могут быть решены в будущих версиях; например, его быстродействие может со временем улучшиться. Даже в существующем на сегодняшний день виде JPython несомненно представляет собой идеальное расширяющее язык решение для приложений, основанных на Java, которое предлагает значительно более законченное решение для управления Java с помощью сценариев, чем те, которые предлагаются другими языками сценариев.[1]

---

[1] Другие языки сценариев решают проблему интеграции с Java путем собственной реализации виртуальной машины Java на соответствующем языке сценариев или путем интегрирования своих первоначальный C-реализаций с Java, используя интерфейс Java машинно-зависимых вызовов. Тот и другой подходы далеки от гладкости и мощи, получаемых при генерации действительного байт-кода Java.

Дополнительные сведения можно найти в пакете Jpython, содержащемся на прилагаемом к книге CD. Кроме того, посетите домашнюю страницу Jpython, которая в настоящее время находится на *http://www.jpython.org*. Просочились также слухи о готовящейся книге по JPython, поэтому следите на *http://www.python. org* за событиями в этой области. Смотрите также ниже врезку относительно новой реализации Python для среды C#/.NET под Windows. Похоже, что скоро можно будет выбирать из трех (а не двух) питонов, а в дальнейшем, может быть, и из большего числа. Все они, вероятно, будут реализовывать один и тот же базовый язык Python, которым мы пользуемся в этой книге, но делать особое ударение на других схемах интеграции, доменах приложений, средах разработчика и т. д.

Grail: веб-броузер на основе Python

Я вкратце упоминал броузер Grail в начале главы 10. Многие из инструментов Python для Интернета используют вложенные затраты и относятся ко времени Grail – развитого броузера Интернета, который:

- Полностью написан на Python
- Для реализации интерфейса пользователя и отображения страниц использует GUI API Tkinter
- Загружает и выполняет сценарии Python/Tkinter как апплеты клиента

Как упоминалось выше, Grail был чем-то вроде проверки идеи использования Python при программировании крупных интернет-приложений. Он реализует все обычные протоколы Интернета и похож в работе на обычные броузеры, такие как Netscape и Internet Explorer. Страницы Grail реализованы с помощью текстовых элементов Tk, знакомых нам по той части этой книги, где описываются GUI.

Более интересно, что броузер Grail позволяет писать на Python апплеты. Апплеты Grail просто представляют собой фрагменты кода Python, располагающиеся на сервере, но выполняющиеся на клиенте. Если документ HTML ссылается на класс и файл Python, находящиеся на машине сервера, Grail автоматически загружает код Python через сокет и выполняет его на машине клиента, передавая ему информацию об интерфейсе пользователя броузера. Загруженный код Python может воспользоваться переданной ему информацией о контексте броузера для настройки интерфейса пользователя, добавления к нему новых графических элементов и осуществления на локальной машине произвольной обработки клиентом. Грубо говоря, апплеты Python в Grail служат тем же целям, что и апплеты Java в обычных броузерах Интернета: они выполняют на стороне клиента задачи, слишком сложные для реализации с помощью других технологий, таких как сценарии CGI на стороне сервера и генерируемый HTML.

Простой пример апплета Grail

Апплеты Grail создаются чрезвычайно просто. На самом деле апплеты суть просто программы Python/Tkinter; за редкими исключениями им вообще не требуется что-либо «знать» о Grail. Рассмотрим короткий пример. Код примера 15.6 просто добавляет в броузер кнопку, которая меняет свой вид при каждом нажатии (ее растровое изображение перенастраивается в обработчике обратного вызова кнопки).

Определение этой страницы состоит из двух частей: файла HTML и кода апплета Python, на который она ссылается. Как обычно, файл *grail.html* описывает, как форматировать веб-страницу, когда в броузере выбран ее URL. Но здесь также есть тег APP, указывающий апплет (класс) Python, который должен выполнить броузер. По умолчанию предполагается, что модуль Python имеет одинаковое с классом имя и нахо-

дится в том же месте (каталоге URL), что и ссылающийся на него файл HTML. Дополнительные параметры тега APP могут переопределять адрес апплета, устанавливаемый по умолчанию.

Пример 15.6. PP2E\Internet\Other\grail.html

```
<HEAD>
<TITLE>Grail Applet Test Page</TITLE>
</HEAD>
<BODY>
<H1>Test an Applet Here!</H1>
Click this button!
<APP CLASS=Question>
</BODY>
```

Файл апплета, на который ссылается HTML, является сценарием Python, добавляющим графические элементы в основанный на Tkinter броузер Grail. Апплеты являются просто классами в модулях Python. Когда в HTML встречается тег APP, броузер Grail загружает модуль с исходным кодом *Question.py* (пример 15.7) и создает экземпляр его класса Question, передавая ему в качестве владельца (родителя) графический элемент броузера. Владелец – это тот крючок, который позволяет апплетам прикреплять новые графические элементы к самому броузеру; таким способом апплеты расширяют GUI, создаваемый HTML.

Пример 15.7. PP2E\Internet\Other\Question.py

```
# Файл апплета Python: Question.py в одном месте (URL) с файлом html,
# который на него ссылается; добавляет элементы в броузер;

from Tkinter import *

class Question:                                              # выполняется из grail?
    def __init__(self, parent):                              # parent=броузер
        self.button = Button(parent, bitmap='question', command=self.action)
        self.button.pack()

    def action(self):
        if self.button['bitmap'] == 'question':
            self.button.config(bitmap='questhead')
        else:
            self.button.config(bitmap='question')

if __name__ == '__main__':
    root = Tk()                                              # выполняется автономно?
    button = Question(root)                                  # parent=Tk: верхний уровень
    root.mainloop()
```

Обратите внимание, что в этом классе нет ничего специфичного для Grail или Интернета. На самом деле его можно выполнить (и протестировать) автономно как программу Python/Tkinter. На рис. 15.3 показано, как выглядит автономное выполнение в Windows (с корневым объектом приложения Tk в качестве родителя); при выполнении в Grail (когда родителем является объект броузер/страница) кнопка возникает как часть веб-страницы. В любом случае картинка на ней изменяется при каждом нажатии.

Рис. 15.3. Автономное выполнение апплета Grail

В сущности, апплеты Grail представляют собой просто модули Python, подключенные к страницам HTML с помощью тега APP. Броузер Grail загружает исходный код, идентифицируемый тегом APP, и выполняет его локально на клиенте во время процесса создания новой страницы. Новые элементы, добавляемые на страницу (как кнопка в этом примере), могут затем выполнять функции Python обратного вызова при активации пользователем.

Апплеты взаимодействуют с пользователем, создавая один или более произвольных графических элементов Tk. Конечно, предыдущий пример является искусственным, но обратите внимание на то, что обработчик обратного вызова этой кнопки может делать все, что можно запрограммировать на Python: обновление постоянно хранящейся информации, показ новых диалоговых окон, вызов расширений C и т. д. Однако при взаимодействии с режимом ограниченного выполнения Python (обсуждаемым далее) можно не позволить апплетам выполнять потенциально небезопасные операции, такие как открытие локальных файлов и связь через сокеты.

На рис. 15.4 показан снимок экрана Grail, дающий представление о возможностях, открывающихся при загрузке кода Python и выполнении его на машине клиента. Там представлена демонстрационная программа «игра в жизнь»; все, что вы здесь видите, реализовано с помощью Python и интерфейса GUI Tkinter. Для запуска демонстрации необходимо установить Python с расширением Tk и загрузить броузер Grail для локального выполнения на машине или копировать его с CD. После этого укажите броузеру URL, где находится какая-нибудь демонстрационная программа Grail.

После всего сказанного я должен добавить, что Grail больше формально не поддерживается и теперь используется главным образом для исследований (Гвидо не намере-

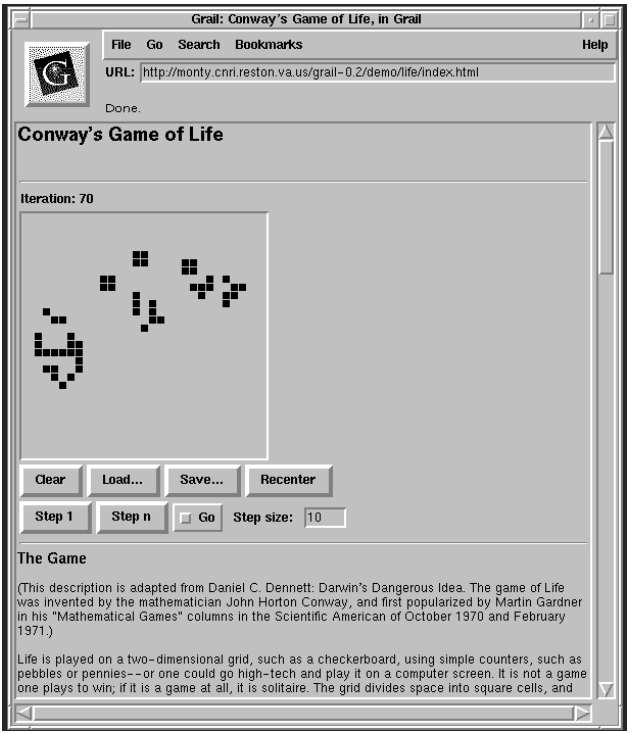


Рис. 15.4. Демонстрация апплета Grail

вался с помощью Grail вывести из игры Netscape или Microsoft). Его по-прежнему можно получить бесплатно (находится на *http://www.python.org*) и использовать для серфинга по веб или экспериментов с альтернативными идеями в веб-броузерах, но это не тот активный проект, каким он был несколько лет назад.

Если вы хотите писать апплеты для веб-броузера на Python, то сегодня чаще используется подход, при котором описывавшаяся выше система JPython компилирует сценарии в файлы байт-кодов апплетов Java и для создания в сценариях интерфейса пользователя используются библиотеки Java. Встраивание кода Python в HTML с помощью расширения Active Scripting, описываемого далее в этой главе, предоставляет еще один способ интеграции кода клиента.

Увы, и этот совет может в будущем оказаться устаревшим. Например, если Tkinter будет перенесен в JPython, можно будет строить GUI в файлах апплетов посредством Tkinter, а не библиотек классов Java. В самом деле, когда я это писал, в Сети по адресу *http://jtkinter.sourceforge.net* имелась предварительная версия полной реализации Java JNI встроенного модуля Python `_tkinter` (позволяющего сценариям JPython импортировать и использовать модуль Tkinter стандартной библиотеки Python). Независимо от того, станет ли в результате Tkinter жизнеспособным вариантом GUI под JPython, все текущие подходы должны измениться. Grail, к примеру, был значительно более известным инструментом, когда писалось первое издание этой книги. Как всегда, следите за развитием событий в сообществе Python на *http://www.python.org*; ясновидение еще не настолько развито, чтобы давать точные прогнозы.

Ограниченный режим выполнения Python

В предыдущих главах я старался подчеркивать опасность выполнения произвольного кода Python, получаемого из Интернета. Ничто не может, например, помешать злоумышленнику послать такую строку, как `os.system('rm *')`, в поле формы, где мы рассчитывали получить просто число; выполнение такой строки кода с помощью встроенной функции `eval` или оператора `exec` по умолчанию может действительно произойти – при этом должны быть удалены все файлы в каталоге сервера или клиента, где выполняется сценарий Python!

Кроме того, действительно злобный пользователь с помощью таких ловушек может просматривать или загружать файлы паролей или иным образом получать доступ, уничтожать или подменять ресурсы вашей машины. Когда есть дырка, тут же появляется хакер. Как я предупреждал, если из формы должно быть получено число, то следует воспользоваться более простыми средствами преобразования строк, таким как `int` или `string.atoi`, а не интерпретировать содержимое поля как синтаксис программ Python с помощью `eval`.

Но что если действительно нужно выполнить код Python, передаваемый через Сеть? Например, вы захотите построить основанную на веб обучающую систему, разрешающую пользователям выполнять код из броузера. Есть возможность сделать это безопасным способом, но для этого потребуется воспользоваться средствами *режима ограниченного выполнения* Python. Поддержка режима ограниченного выполнения обеспечивается двумя стандартными библиотечными модулями, `rexec` и `bastion`. `rexec` служит главным интерфейсом ограниченного выполнения, а `bastion` можно использовать для ограничения и контроля доступа к атрибутам объектов.

В Unix-системах можно также использовать стандартный модуль resource, чтобы ограничить при выполнении кода такие вещи, как время использования CPU и расход памяти. В руководстве по библиотеке Python эти модули подробно обсуждаются, но мы кратко рассмотрим здесь rexec.

Использование rexec

Ограниченный режим исполнения, реализуемый rexec, является необязательным – по умолчанию весь код Python выполняется с полным доступом ко всему, что имеется в языке Python и библиотеке. Но при включении ограниченного режима код выполняется в модели так называемой «песочницы» – доступ к компонентам локальной машины ограничивается. Потенциально небезопасные операции либо запрещаются, либо должны быть подтверждены кодом, который можно настраивать путем создания подклассов. Например, сценарий примера 15.8 выполняет строку программного кода в ограниченном режиме и настраивает класс rexec так, чтобы ограничить доступ к файлам одного специфического каталога.

Пример 15.8. PP2E\Internet\Other\restricted.py

```
#!/usr/bin/python
import rexec, sys
Test = 1
if sys.platform[:3] == 'win':
    SafeDir = r'C:\temp'
else:
    SafeDir = '/tmp/'

def commandLine(prompt='Input (ctrl+z=end) => '):
    input = ''
    while 1:
        try:
            input = input + raw_input(prompt) + '\n'
        except EOFError:
            break
    print # clear for Windows
    return input

if not Test:
    import cgi                          # выполняется в Сети? - код из формы
    form  = cgi.FieldStorage()          # или интерактивный ввод для проверки
    input = form['input'].value
else:
    input = commandLine()

# создание подкласса для настройки правил по умолчанию: default=режимы записи запрещены
class Guard(rexec.RExec):
    def r_open(self, name, mode='r', bufsz=-1):
        if name[:len(SafeDir)] != SafeDir:
            raise SystemError, 'files outside %s prohibited' % SafeDir
        else:
            return open(name, mode, bufsz)

# ограничить системные ресурсы (не действует в Windows)
if sys.platform[:3] != 'win':
    import resource                     # не более 5 секунд cpu
    resource.setrlimit(resource.RLIMIT_CPU, (5, 5))

# безопасное выполнение строки кода
```

```
guard = Guard()
guard.r_exec(input)      # назначить guard ответственным за проверки и операции открытия файлов
```

При выполнении с помощью этого сценария строк кода Python в Windows безопасный код работает как обычно, и можно читать и записывать файлы, находящиеся в каталоге *C:\temp*, потому что метод `r_open` пользовательского класса `Guard` разрешает работу с файлами, имена которых начинаются с «C:\temp». Действующий по умолчанию `r_open` в `rexec.RExec` разрешает чтение всех файлов, но отказывает всем запросам записи. Для проверки введем код интерактивно, хотя при получении этой строки через Интернет в поле формы сценария CGI происходит в точности то же самое:

```
C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => x = 5
Input (ctrl+z=end) => for i in range(x): print 'hello%d' % i,
Input (ctrl+z=end) => hello0 hello1 hello2 hello3 hello4

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => open(r'C:\temp\rexec.txt', 'w').write('Hello rexec\n')
Input (ctrl+z=end) =>

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => print open(r'C:\temp\rexec.txt', 'r').read()
Input (ctrl+z=end) => Hello rexec
```

С другой стороны, наш пользовательский класс откажет в доступе к файлам за пределами разрешенного каталога, как и при попытке небезопасных по своей сути действий, таких как открытие сокетов, что `rexec` по умолчанию всегда отвергает:

```
C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => open(r'C:\stuff\mark\hack.txt', 'w').write('BadStuff\n')
Input (ctrl+z=end) => Traceback (innermost last):
  File "restricted.py", line 41, in ?
    guard.r_exec(input)                    # пусть guard проверит и выполнит открытие
  File "C:\Program Files\Python\Lib\rexec.py", line 253, in r_exec
    exec code in m.__dict__
  File "<string>", line 1, in ?
  File "restricted.py", line 30, in r_open
    raise SystemError, 'files outside %s prohibited' % SafeDir
SystemError: files outside C:\temp prohibited

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => open(r'C:\stuff\mark\secret.py', 'r').read()
Input (ctrl+z=end) => Traceback (innermost last):
  File "restricted.py", line 41, in ?
    guard.r_exec(input)                    # пусть guard проверит и выполнит открытие
  File "C:\Program Files\Python\Lib\rexec.py", line 253, in r_exec
    exec code in m.__dict__
  File "<string>", line 1, in ?
  File "restricted.py", line 30, in r_open
    raise SystemError, 'files outside %s prohibited' % SafeDir
SystemError: files outside C:\temp prohibited

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => from socket import *
Input (ctrl+z=end) => s = socket(AF_INET, SOCK_STREAM)
Input (ctrl+z=end) => Traceback (innermost last):
  File "restricted.py", line 41, in ?
    guard.r_exec(input)                    # пусть guard проверит и выполнит открытие
  ...part ommitted...
```

```
  File "C:\Program Files\Python\Lib\ihooks.py", line 324, in load_module
    exec code in m.__dict__
  File "C:\Program Files\Python\Lib\plat-win\socket.py", line 17, in ?
    _realsocketcall = socket
NameError: socket
```

А что со страшной проблемой `rm *`? Этот код можно выполнить в обычном режиме Python, как и все другое, но не при работе в ограниченном режиме. В режиме ограниченного выполнения Python отключает некоторые потенциально опасные атрибуты модуля `os`, такие как `system` (для выполнения команд оболочки):

```
C:\temp>python
>>> import os
>>> os.system('ls -l rexec.txt')
-rwxrwxrwa   1 0        0             13 May  4 15:45 rexec.txt
0
>>>
C:\temp>python %X%\Part2\internet\other\restricted.py
Input (ctrl+z=end) => import os
Input (ctrl+z=end) => os.system('rm *.*')
Input (ctrl+z=end) => Traceback (innermost last):
  File "C:\PP2ndEd\examples\Part2\internet\other\restricted.py", line 41, in ?
    guard.r_exec(input)      # ask guard to check and do opens
  File "C:\Program Files\Python\Lib\rexec.py", line 253, in r_exec
    exec code in m.__dict__
  File "<string>", line 2, in ?
AttributeError: system
```

Внутренне режим ограниченного выполнения действует, лишая доступа к некоторым API (например, контролируется импорт) и изменяя словарь `__builtins__` в том модуле, где выполняется ограниченный в правах код, чтобы ссылки указывали на специальную и безопасную версию пространства встроенных имен стандартного `__builtin__`. Например, специальная версия `__builtins__.open` ссылается на ограниченную версию стандартной функции открытия файлов. `rexec` также хранит настраиваемые списки безопасных встроенных модулей, безопасных атрибутов модулей `os` и `sys`, и др. Подробнее об этом читайте в руководстве по библиотеке Python.

Режим ограниченного выполнения не обязательно связан со сценариями. Им полезно воспользоваться, когда требуется выполнить код Python сомнительного происхождения. Например, далее в этой книге в программе калькулятора мы будем использовать встроенные функции Python `eval` и `exec` для вычисления арифметических выражений и ввода команд. Так как в таком контексте данные, вводимые пользователем, вычисляются как исполняемый код, ничто не мешает пользователю по неведению или умышленно ввести код, выполнение которого вызовет разрушения (например, при случайном вводе кода Python, удаляющего файлы). Однако риск выполнения строк с исходным кодом преобладает в приложениях, выполняемых в веб, так как они по своей сущности открыты как для пользы, так и для злоупотребления. Хотя JPython наследует базовую модель безопасности Java, системам с чистым Python, таким как Zope, Grail и пользовательским сценариям CGI, полезно выполнять передаваемые через Сеть строки в ограниченном режиме.

Средства обработки XML

Python поставляется вместе с поддержкой синтаксического анализа XML в стандартной библиотеке, и у него есть энергичная группа пользователей, специализирующаяся на XML. *XML* (eXtended Markup Language) представляет собой основанный на тегах язык разметки для описания различного рода структурированных данных. Среди прочих употреблений, XML принят многими компаниями как средство стандартного представления контента в базах данных и Интернете. Будучи объектно-ориентированным языком сценариев, Python очень хорошо сочетается с базовой идеей XML-обмена структурированными документами и может стать ведущим игроком в области XML.

XML основан на синтаксисе тегов, знакомом авторам веб-страниц. Модуль библиотеки Python `xmllib` содержит средства для синтаксического анализа XML. Если коротко, то для использования этого анализатора XML определяется подкласс класса Python `XMLParser`, и его методы служат функциями обратного вызова, вызываемыми при обнаружении различных структур XML. Анализ текста в значительной мере автоматизирован библиотечным модулем. В конце исходного кода этого модуля в файле *xmllib.py* библиотеки Python содержится код самотестирования, сообщающий дополнительные детали относительно его применения. Вместе с Python поставляется также стандартный анализатор HTML, `htmllib`, действующий на аналогичных принципах и основанный на модуле анализатора SGML `sgmllib`.

К сожалению, поддержка XML в Python все еще развивается, и описание ее выходит далеко за рамки этой книги. Вместо более подробного изложения я направлю вас к источникам дополнительной информации:

Стандартная библиотека

Прежде всего, следует обратиться к руководству по библиотеке Python за дополнительными сведениями о средствах поддержки XML в стандартной библиотеке. В настоящее время они состоят только из модуля анализатора `xmllib`, но в дальнейшем их может стать больше.

Пакет PyXML SIG

На момент написания этой книги лучшим источником по инструментам Python для работы с XML и документации к ним служит веб-страница XML SIG (Special Interest Group – группа особого интереса) на *http://www.python.org* (щелкните вверху по ссылке «SIGs»). Эта группа работает над соединением технологий XML с Python и предоставляет дистрибутив пакета инструментов XML под названием *PyXML*. В этот пакет входят средства, еще не включенные в стандартный дистрибутив Python, в том числе анализаторы XML, реализованные на C и Python, реализация на Python SAX и DOM (объектная модель документов XML), интерфейс Python к анализатору `Expat`, примеры кода, документация и тестовый комплект.

Средства сторонних разработчиков

Можно также найти бесплатные средства Python для поддержки XML сторонних разработчиков в веб по ссылкам, имеющимся на веб-странице XML SIGs. В их число входит DOM-реализация для сред CORBA (4DOM), поддерживающая в настоящая время два ORB (ILU и Fnorb), и многое другое.

Документация

Я пишу эти слова накануне опубликования книги, посвященной обработке XML с помощью Python; подробности ищите в списке книг на *http://www.python.org* или в своем любимом книжном магазине.

С учетом быстрого развития XML-технологий я не поручусь, что через несколько лет после выхода этого издания какие-либо из указанных ресурсов сохранят актуальность, поэтому следите за последними событиями в этой области на сайте Python.

На самом деле история с XML существенно изменилась в промежуток времени между написанием этой книги и окончательной сдачей ее в O'Reilly. В Python 2.0 некоторые из инструментов, описываемых здесь как входящие в пакет PyXML SIG, попали в пакет стандартного модуля `xml` в библиотеке Python. Иными словами, они поставляются и устанавливаются вместе с самим Python; подробности смотрите в руководстве по библиотеке Python 2.0. У O'Reilly находится в работе книга по этой теме под названием *Python и XML*.

Расширения для веб-сценариев в Windows

В данной книге не рассказывается в подробностях о специфических для Windows расширениях Python, но здесь следует бросить беглый взгляд на средства создания интернет-сценариев, которые есть у программистов Windows. В Windows Python можно использовать как язык сценариев для систем Active Scripting и Active Server Pages, с помощью которых осуществляется управление основанными на HTML клиентскими и серверными приложениями. В более общем смысле программы Python могут играть роль клиентов и серверов COM и DCOM под Windows.

В данный момент вы должны отметить себе, что все описываемое в этом разделе работает только с инструментами Microsoft, и встраивание в HTML действует только в Internet Explorer (у клиента) и Internet Information Server (на сервере). Если вам нужна переносимость, ваши проблемы лучше решать с помощью других систем, описанных в этой главе (см. апплеты для клиентов JPython, поддержку сценариев сервера в PSP и модель Zope публикации объектов на сервере). Напротив, если переносимость не играет роли, описываемая ниже техника дает мощные средства создания сценариев для обеих сторон, общающихся через веб.

Active Scripting: встраивание на стороне клиента

Active Scripting представляет собой технологию для связи языков сценариев с ведущими приложениями (hosting application). Ведущее приложение предоставляет специфический для него *API объектной модели*, открывающий доступ к объектам и функциям, которые можно использовать в программах языка сценариев.

В одном из наиболее частых своих применений система Active Scripting предоставляет средства, позволяющие коду языка сценариев, встроенному в страницы HTML, обмениваться данными с локальным веб-сервером посредством автоматически предоставляемого API объектной модели. Internet Explorer, например, с помощью Active Scripting экспортирует глобальные функции и объекты интерфейса пользователя, используемые в сценариях, встраиваемых в HTML. С помощью Active Scripting код Python может встраиваться в HTML веб-страницы, будучи заключен в специальные теги. Такой код выполняется на машине клиента и играет ту же роль, что и встроенный JavaScript и VBScript.

Основы Active Scripting

К сожалению, встраивание Python в HTML клиента функционирует только на машинах, где установлен Python и Internet Explorer настроен на взаимодействие с языком

Python (в результате установки пакета расширения win32all, о чем будет сказано чуть ниже). Поэтому данная технология не применима с большинством броузеров, имеющихся сегодня в киберпространстве. С другой стороны, если есть возможность конфигурировать машины, на которые должна быть установлена система, это не составляет проблемы.

Прежде чем рассматривать пример Python, посмотрим, каким образом стандартные броузеры работают с языками, встраиваемыми в HTML. По умолчанию IE (Internet Explorer) распознает JavaScript (на самом деле Jscript – его реализацию Microsoft) и VBScript (производный от Visual Basic), поэтому оба эти языка можно встраивать в любую поставляемую систему. Например, файл HTML примера 15.9 содержит встроенный код JavaScript, установленный на моем PC в качестве языка сценариев IE по умолчанию.

Пример 15.9. PP2E\Internet\Other\activescript-js.html

```
<HTML>
<BODY>
<H1>Embedded code demo: JavaScript</H1>
<SCRIPT>

// popup 3 alert boxes while this page is
// being constructed on client side by IE;
// JavaScript is the default script language,
// and alert is an automatically exposed name

function message(i) {
    if (i == 2) {
        alert("Finished!");
    }
    else {
        alert("A JavaScript-generated alert => " + i);
    }
}

for (count = 0; count < 3; count += 1) {
    message(count);
}

</SCRIPT>
</BODY></HTML>
```

Текст этого файла между тегами <SCRIPT> и </SCRIPT> представляет собой код JavaScript. Не тревожьтесь по поводу его синтаксиса – эта книга не о JavaScript, а более простой эквивалент Python будет приведен ниже. Важно понять, как броузер обрабатывает этот код.

Если броузер обнаруживает такой блок кода во время построения новой страницы, он находит соответствующий интерпретатор, сообщает интерпретатору глобальные имена объектов и передает код на выполнение интерпретатору. Глобальные имена становятся переменными во встроенном коде и обеспечивают связь с контекстом броузера. Например, имя alert в блоке кода ссылается на глобальную функцию, которая создает окно сообщения. Другие глобальные имена ссылаются на объекты, предоставляющие доступ к интерфейсу пользователя броузера: объекты окон, объекты документов и т. д.

Этот файл HTML можно запустить на локальной машине, щелкнув по его имени в менеджере файлов. Его можно также записать на удаленный сервер и обращаться к нему в броузере по URL. Каким бы способом вы его ни запустили, при построении страницы всплывают три окна с предупреждениями, созданные встроенным кодом. На рис. 15.5 показано одно такое окно в IE.

Рис. 15.5. IE, выполняющий встроенный код JavaScript

Версия этого примера с VBScript (Visual Basic) представлена в примере 15.10, чтобы вы могли их сравнить.[1] При ее выполнении создаются три аналогичных окна, но в них всюду написано: «VBScript». Обратите внимание на параметр `Language` в теге `<SCRIPT>`; с его помощью необходимо объявить язык для IE (в данном случае VBScript), если язык встроенных сценариев отличается от языка по умолчанию. В версии JavaScript примера 15.9 `Language` не требовался, потому что JavaScript был языком по умолчанию. За исключением этого объявления, IE безразлично, какой язык вставлен между тегами `<SCRIPT>` и `</SCRIPT>`; в принципе, Active Scripting является безразличным к языку механизмом сценариев.

Пример 15.10. PP2E\Internet\Other\activescript-vb.html

```
<HTML>
<BODY>
<H1>Embedded code demo: VBScript</H1>
<SCRIPT Language=VBScript>

' do the same but with embedded VBScript;
' the Language option in the SCRIPT tag
' tells IE which interpreter to use

sub message(i)
    if i = 2 then
        alert("Finished!")
    else
        alert("A VBScript-generated alert => " & i)
    end if
end sub
```

[1] Опять же, можете не обращать внимания на синтаксис этого примера. В этой книге я не собираюсь учить синтаксису JavaScript или VBScript, так же как не скажу, какая из трех версий этого примера более понятная (хотя вы, вероятно, догадываетесь о моих предпочтениях). Первые две версии включены частично для того, чтобы их могли сравнить читатели, имеющие опыт разработок для веб.

```
for count = 0 to 2 step 1
    message(count)
next

</SCRIPT>
</BODY></HTML>
```

А как же поместить на эту страницу код Python? К сожалению, сначала нужно сделать кое-что еще. Хотя IE, в принципе, нейтрален в отношении языка, он все же поддерживает одни языки лучше, чем другие, по крайней мере, сегодня. Кроме того, другие броузеры могут быть более строгими и не поддерживать концепцию Active Scripting вовсе.

Например, на моей машине с установленными у меня версиями броузеров (IE 5, Netscape 4) предшествующий пример JavaScript работает как в IE, так и в Netscape, но версия Visual Basic работает только в IE. То есть IE непосредственно поддерживает VBScript и JavaScript, тогда как Netscape обрабатывает только JavaScript. Ни один из броузеров в том виде, как он установлен, не может выполнять встроенный код Python несмотря на то, что сам Python установлен на моей машине. Нужны дополнительные действия, которые позволят заменить код JavaScript и VBScript, встроенный в наши страницы HTML, на Python.

Учим IE обрабатывать Python

Чтобы заставить работать версию с Python, недостаточно просто установить Python на своем PC. Необходимо также зарегистрировать Python в IE. К счастью, это делается почти автоматически благодаря разработчикам расширений Python для Windows; необходимо просто установить пакет расширений Windows.

Вот как это делается. Средство, осуществляющее регистрацию, входит в состав расширений Python Win32, не включенных в стандартную систему Python. Чтобы IE узнал о Python, необходимо сделать следующее:

1. Сначала установить на PC стандартный дистрибутив Python (вы уже должны были сделать это к настоящему времени – просто дважды щелкнуть по программе самоинсталляции Python).
2. Отдельно загрузить с *http://www. python.org* и установить пакет win32all (его можно также взять с прилагающегося к книге CD).[1]

Пакет `win32all` содержит расширения для Python `win32COM` и *PythonWin* IDE (простой интерфейс для редактирования и запуска программ Python, написанных с интерфейсами MFC в `win32all`), а также массу других специфических для Windows средств, о которых не рассказывается в этой книге. К делу здесь относится то, что при установке `win32all` происходит автоматическая регистрация Python для использования в IE. При необходимости можно осуществить регистрацию вручную, выполнив сценарий Python *python\win32comext\axscript\client\pyscript.py*, находящийся в пакете расширений win32.

После того как Python зарегистрирован таким способом в IE, код Python, встроенный в HTML, работает так же, как наши примеры JavaScript и VBScript – IE устанавливает глобальные имена Python, чтобы сделать доступным свою объектную модель, и передает встроенный код на исполнение интерпретатору Python. Пример 15.11 снова показывает наши окна предупреждений, создаваемые встроенным кодом Python.

---

[1] Однако пакет `win32all` может и не потребоваться. Например, дистрибутив ActivePython, поставляемый ActiveState (*http://www.activestate.com*), содержит пакет расширений для Windows.

Пример 15.11. PP2E\Internet\Other\activescript-py.html

```
<HTML>
<BODY>
<H1>Embedded code demo: Python</H1>
<SCRIPT Language=Python>

# Делает все то же, на на Python.
# Встроенный код отображает три всплывающих окна после закгрузки страницы.
# Любой код Python будет здесь работать. Возможно использование автоматически-импортируемых
# глобальных функций и объектов.

def message(i):
    if i == 2:
        alert("Finished!")
    else:
        alert("A Python-generated alert => %d" % i)

for count in range(3): message(count)

</SCRIPT>
</BODY></HTML>
```

На рис. 15.6 показано одно из трех всплывающих окон, которые вы должны увидеть при открытии этого файла в IE после установки пакета `win32all` (для его открытия можно просто щелкнуть по его значку в менеджере файлов Windows). Заметьте, что при первом обращении к этой странице IE может потребоваться загрузить Python, что на медленных машинах повлечет заметную задержку; последующие обращения обычно происходят значительно быстрее, потому что Python уже загружен.

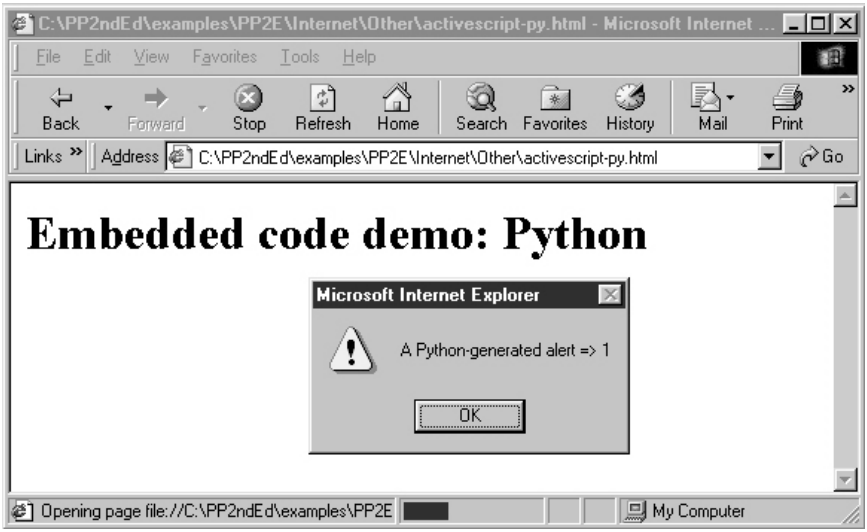


Рис. 15.6. IE, выполняющий встроенный код Python

К сожалению, этот пример на моей машине все-таки работает только в IE версии 4 и выше, но не в броузере Netscape (и согласно имеющимся сведениям не работает также в Netscape 6 и Mozilla). Иными словами (по крайней мере, сегодня и исключая новые версии броузеров), Active Scripting не только является технологией, действующей только в Windows, но как инструмент веб-броузера клиента для Python действует только на машинах, где установлен и зарегистрирован в IE Python, и даже там – только с IE.

Следует также знать, что даже если заставить Python работать в IE, код Python работает в *режиме ограниченного выполнения*, с ограниченным доступом к ресурсам машины (например, ваш код не сможет открывать сокеты – смотрите подробности в описании модуля `rexec` выше в этой главе). По-видимому, это и требуется при выполнении кода, загружаемого через Сеть, и может быть изменено локально (реализация закодирована в Python), но ставит границы полезности и области применения сценариев Python, встраиваемых в HTML.

Но хорошо, что это все-таки действует – после простой настройки код Python можно встраивать в HTML и заставить выполняться в IE так же, как JavaScript и VBScript. Во многих приложениях ограничение на использование только Windows и IE совершенно приемлемо. Active Scripting является простым способом добавить клиентские сценарии Python в веб-броузеры, особенно если есть возможность управлять средой, для которой поставляется система. Например, машины в интрасетях компаний могут иметь хорошо известные конфигурации. В таких ситуациях Active Scripting дает возможность разработчикам применить всю мощь Python в сценариях для клиентов.

Active Server Pages: встраивание кода в сценарии сервера

Active Server Pages (ASP) использует сходную модель: код Python встраивается в HTML, описывающий веб-страницу. Но ASP является технологией *сервера*; встроенный код Python выполняется на машине сервера и использует основанный на объектах API для динамической генерации фрагментов HTML, который в итоге отправляется броузеру клиента. Как мы видели в последних трех главах, CGI-сценарии Python на сервере встраивают и генерируют HTML и действуют с необработанными входными данными и выходными потоками. Напротив, сценарии ASP для серверов встроены в HTML и решают свои задачи с помощью объектной модели более высокого уровня.

Так же как Active Scripting на стороне клиента, ASP требует установки Python и пакета расширений Python для Windows `win32all`. Но так как ASP выполняет встроенный код на сервере, конфигурировать Python нужно только на одной машине. Как и сценарии CGI в целом, сценарии Python ASP обычно имеют значительно более широкую область применения, так как не требуется поддержки Python на каждом клиенте. Однако в отличие от сценариев CGI для ASP в настоящее время требуется использование Microsoft IIS (Internet Information Server).

Короткий пример ASP

Мы не имеем здесь возможности сколь-нибудь подробно обсуждать ASP, но приведем пример ASP-файла со встроенным кодом Python:

```
<HTML><BODY>
<SCRIPT RunAt=Server Language=Python>
#
# находящийся здесь код выполняется на сервере
#
</SCRIPT>
</BODY></HTML>
```

Как и прежде, код помещается внутрь пары тегов `SCRIPT`. На этот раз с помощью параметра `RunAt` мы сообщаем ASP о необходимости выполнить код на сервере. Если опустить этот параметр, то код с тегами будет передан клиенту и выполнен в IE (если тот правильно настроен). ASP распознает также код, заключенный между разделителями `<%` и `%>`, и позволяет задать язык для всей страницы. Такой формат удобнее, если на странице есть несколько фрагментов кода, как в примере 15.12.

Пример 15.12. PP2E\Internet\Other\asp-py.asp

```
<HTML><BODY>
<%@ Language=Python %>

<%
#
# Здесь код Python, использующий глобальные имена Request (ввод), Response (вывод) и т. д.
#
Response.Write("Hello ASP World from URL %s" %
                          Request.ServerVariables("PATH_INFO"))
%>
</BODY></HTML>
```

Как бы ни был помечен код, ASP выполняет его на сервере, передав в него ряд именованных объектов, посредством которых код может осуществлять доступ к входным и выходным данным и контексту сервера. Например, автоматически импортируемые объекты `Request` и `Response` дают доступ к контексту ввода и вывода. Здесь код вызывает метод `Response.Write`, чтобы послать текст в броузер клиента (подобно оператору print в обычном CGI-сценарии Python), а также метод `Request.ServerVariables`, чтобы получить информацию переменных окружения. Чтобы заставить этот сценарий выполняться живьем, нужно поместить его в надлежащий каталог на машине сервера, выполняющей IIS с поддержкой ASP.

Соединение через COM

В своей основе как IE, так и IIS основываются на системе интеграции COM (Component Object Model, модель многокомпонентных объектов), они реализуют свои объектные API с помощью стандартных интерфейсов COM и видны со стороны, как любые другие COM-объекты. В более широком плане Python может быть использован как язык сценариев и как язык реализации для любого COM-объекта. Хотя механизм COM, применяемый для выполнения кода Python, встроенного в HTML, автоматизирован и скрыт, его можно применять явно, чтобы программы Python выполняли роль клиентов и серверов COM. COM является общей технологией интеграции и не привязан лишь к созданию сценариев для Интернета, но краткое знакомство здесь с этой моделью может в некоторой мере снять таинственность с магии Active Scripting, на которой основано встраивание кода в HTML.

Краткое введение в COM

COM является технологией Microsoft для независимой от языков интеграции компонентов. Иногда она выступает под коммерческим названием ActiveX, отчасти является производной от системы под названием OLE и служит технологической основой системы Active Scripting, с которой мы познакомились выше.[1] COM предоставляет также расширение для распределенных систем, известное как DCOM и позволяющее связываться между собой объектам, выполняющимся на удаленных машинах. Реализация DCOM часто заключается просто в ряде настроек реестра Windows, позволяющих связать серверы с машинами, на которых они выполняются.

---

[1] Можно сказать, что OLE (Object Linking and Embedding) является предтечей COM, а Active Scripting является лишь технологией, определяющей COM-интерфейсы для таких действий, как передача объектов по имени интерпретаторам произвольных языков программирования. Active Scripting несильно выходит за рамки собственно COM, предоставляя лишь немногие расширения, но перегруженность акронимами и модными словечками пышно расцвела в разработке ПО для Windows.

Функционально COM определяет стандартный способ, при котором объекты, реализованные на произвольных языках, общаются друг с другом с помощью опубликованной объектной модели. Например, компоненты COM могут быть написаны на Visual Basic, Visual C++, Delphi, PowerBuilder и Python и использоваться программами, написанными с помощью этих систем. Благодаря тому, что слой косвенной адресации COM скрывает различия между всеми участвующими языками, существует, например, возможность использовать в Visual Basic объект, реализованный на Python, и наоборот.

Кроме того, многие программные пакеты регистрируют интерфейсы COM, поддерживающие сценарии конечных пользователей. Например, Microsoft Excel публикует объектную модель, позволяющую любому языку сценариев с поддержкой COM запустить Excel и получить программный доступ к данным электронных таблиц. Аналогично с помощью COM можно управлять Microsoft Word и автоматически обрабатывать документы. Независимость COM от языка означает, что программы, написанные на любом языке программирования с COM-интерфейсом, в том числе Visual Basic и Python, могут автоматизировать работу Excel и Word.

Для данной главы основным является то, что Active Scripting тоже предоставляет объекты COM, позволяющие встраиваемым в HTML сценариям связываться с Microsoft Internet Explorer (у клиента) и Internet Information Server (на сервере). Обе системы регистрируют свои объектные модели в Windows, благодаря чему их можно вызывать из любого языка, поддерживающего COM. Например, когда Internet Explorer извлекает и выполняет код Python, встроенный в HTML, имена некоторых переменных Python автоматически устанавливаются соответствующими объектным компонентам COM, предоставляющим доступ к контексту и средствам IE (`alert` в примере 5.11). Обращения к таким компонентам из кода Python автоматически направляются через COM в IE.

Клиенты COM в Python

Установка пакета расширения Python `win32all` позволяет создавать программы Python, служащие зарегистрированными клиентами и серверами COM и не имеющие вообще никакого отношения к Интернету. Скажем, программа Python в примере 15.13 действует как клиент COM-объекта Microsoft Word.

Пример 15.13. PP2E\Internet\Other\Com\comclient.py

```
##############################################################################
# Клиент COM, написанный на Python: связь с MS-Word через его объектную модель COM;
# использует динамическую диспетчеризацию (поиск/связывание во время выполнения)
# или статическую и более быструю с библиотекой типов, если был выполнен makepy.py;
# для пользования этим интерфейсом нужно установить пакет расширения
# win32all; Word выполняется скрыто, если не установить Visible=1
# (Visible позволяет наблюдать, но влияет на интерактивные сеансы Word);
##############################################################################

from sys import argv
docdir = 'C:\\temp\\'
if len(argv) == 2: docdir = argv[1]             # например: comclient.py a:\

from win32com.client import Dispatch            # раннее или позднее связывание
word  = Dispatch('Word.Application')            # соединиться/запустить word
word.Visible = 1                                # иначе word выполняется невидимо

# создать и сохранить новый файл документа
newdoc = word.Documents.Add()                   # вызов методов word
spot = newdoc.Range(0,0)
spot.InsertBefore('Hello COM client world!')    # вставить текст
newdoc.SaveAs(docdir + 'pycom.doc')             # сохранить в файле .doc
```

```
newdoc.SaveAs(docdir + 'copy.doc')
newdoc.Close()

# открыть и изменить файл документа
olddoc = word.Documents.Open(docdir + 'copy.doc')
finder = word.Selection.Find
finder.text = 'COM'
finder.Execute()
word.Selection.TypeText('Automation')
olddoc.Close()

# and so on: see Word's COM interface specs
```

Этот сценарий при необходимости запускает Microsoft Word, который сценариям-клиентам известен как `Word.Application`, и общается с ним через COM. Это значит, что вызовы в этом сценарии автоматически передаются из Python в Microsoft Word и обратно. Этот код существенно использует вызовы, экспортируемые Word, которые не описываются в данной книге. Однако, вооружившись документацией по API объекта Word, можно с помощью таких вызовов писать сценарии Python, автоматизирующие обновление документов, вставляющие и заменяющие текст, создающие и печатающие документы и т. д.

Например, на рис. 15.7 показаны два *.doc*-файла Word, сгенерированные при выполнении предыдущего сценария в Windows: оба являются новыми файлами, и один представляет собой копию другого, в которой осуществлена замена текста. Взаимодействие, происходящее при выполнении этого сценария, еще интереснее: так как атрибут Word `Visible` установлен в 1, можно действительно *наблюдать*, как Word вставляет и заменяет текст, сохраняет файлы и т. д. в ответ на вызовы из сценария. (К сожалению, я не смог сообразить, как вставить в эту книгу видеоклип.)

Вообще говоря, вызовы COM-клиента Python могут осуществляться динамически путем поиска в реестре Windows во время выполнения или статически с помощью библиотек типов, создаваемых при выполнении вспомогательного сценария Python (*makepy.py*) во время разработки. Эти режимы диспетчеризации иногда называют *позд-*

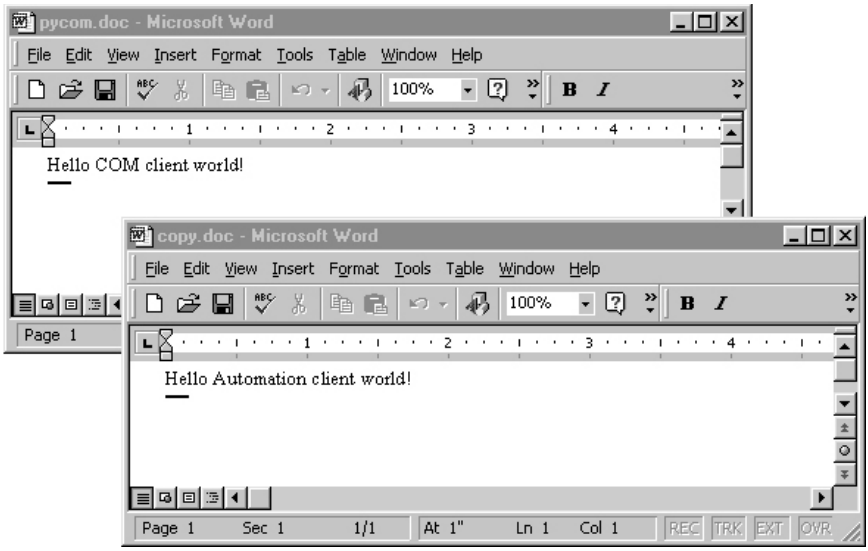


Рис. 15.7. Файлы Word, генерируемые клиентом COM Python

ним и *ранним* связыванием соответственно. При динамическом (позднем) связывании пропускается один этап разработки, но выполнение клиентов происходит медленнее из-за необходимости поиска.[1]

К счастью, при написании сценариев клиентов не требуется знать, какая из схем будет использована. Вызов `Dispatch`, примененный в примере 15.13 для соединения с Word, самостоятельно выберет статическое связывание, если существует библиотека типов сервера, и динамическое связывание в противном случае. Для принудительного использования позднего связывания и игнорирования имеющихся библиотек типов нужно заменить первую строку следующей:

```
from win32com.client.dynamic import Dispatch     # всегда позднее связывание
```

В любой модели вызовов COM-интерфейс Python выполнит все действия по поиску указанного сервера, нахождению и вызову требуемых методов или атрибутов и при необходимости преобразованию типов данных Python согласно стандартному соответствию типов. В контексте Active Scripting базовая модель COM действует таким же образом, но сервером служит IE или IIS (а не Word), набор доступных вызовов другой, и некоторым переменным Python предварительно присваиваются объекты COM-сервера. Понятия «клиент» и «сервер» в таких ситуациях становятся несколько расплывчатыми, но итоговые результаты аналогичны.

Серверы COM в Python

Сценарии Python можно также размещать как *серверы* COM, предоставляя методы и атрибуты, доступные любым языкам программирования или системам, поддерживающим COM. Эта тема слишком сложна, чтобы ее можно было здесь достаточно хорошо осветить, но экспорт объекта Python в COM в основном состоит в том, чтобы обеспечить набор атрибутов класса, идентифицирующих сервер, и выполнить правильные вызовы утилиты регистрации из `win32com`. В примере 15.14 представлен простой COM-сервер, написанный на Python в виде класса.

Пример 15.14. PP2E\Internet\Other\Com\comserver.py

```
###############################################################################################
# Сервер COM, написанный на Python; атрибуты класса _reg_ задают параметры регистрации,
# а остальные перечисляют методы и атрибуты. Для работы этого сценария нужно установить
# Python и пакет win32all, файл этого модуля должен находиться в пути поиска Python,
# а сервер должен быть зарегистрирован в COM (см. код в конце); для создания
# собственного ключа _reg_clsid_ выполните pythoncom.CreateGuid();
###############################################################################################

import sys
from   win32com.server.exception import COMException         # что возбуждать
import win32com.server.util                                  # серверные инструменты
globhellos = 0

class MyServer:

    # настройки данных com
    _reg_clsid_      = '{1BA63CC0-7CF8-11D4-98D8-BB74DD3DDE3C}'
    _reg_desc_       = 'Example Python Server'
    _reg_progid_     = 'PythonServers.MyServer'              # внешнее имя
    _reg_class_spec_ = 'comserver.MyServer'                  # внутреннее имя
```

[1] В действительности `makepy` теперь тоже можно выполнить на этапе прогона, поэтому больше не нужно выполнять его вручную на этапе разработки. Смотрите самые свежие подробности в документации по `makepy`, имеющейся в последнем пакете расширений Windows.

```
    _public_methods_ = ['Hello', 'Square']
    _public_attrs_   = ['version']

    # методы python
    def __init__(self):
        self.version = 1.0
        self.hellos  = 0
    def Square(self, arg):                                  # экспортируемые методы
        return arg ** 2
    def Hello(self):                                        # глобальные переменные
        global globhellos                                   # хранят состояние, но
        globhellos  = globhellos  + 1                       # переменные self - нет
        self.hellos = self.hellos + 1
        return 'Hello COM server world [%d, %d]' % (globhellos, self.hellos)

# функции регистрации
def Register(pyclass=MyServer):
    from win32com.server.register import UseCommandLine
    UseCommandLine(pyclass)
def Unregister(classid=MyServer._reg_clsid_):
    from win32com.server.register import UnregisterServer
    UnregisterServer(classid)

if __name__ == '__main__':    # зарегистрировать сервер при запуске файла
    Register()                # отмена регистрации по аргументу командной строки --unregister
```

Как обычно, этот файл Python нужно поместить в каталог, присутствующий в пути поиска модулей Python. Кроме самого класса сервера, в файле есть код, автоматически проводящий регистрацию и отмену регистрации сервера в COM при запуске файла:

- Для *регистрации* сервера нужно вызвать функцию `UseCommandLine` в пакете `win32com.server.register` и передать ей класс сервера Python. Эта функция использует значения специальных атрибутов класса, чтобы сделать сервер известным в COM. Файл устроен так, что автоматически вызывает средства регистрации при автономном запуске (например, при щелчке в менеджере файлов).
- Для *отмены регистрации* сервера нужно передать при запуске этого файла аргумент командной строки `--unregister`. В этом случае сценарий автоматически снова вызывает `UseCommandLine`, чтобы разрегистрировать сервер; в соответствии со своим именем эта функция изучает аргументы командной строки и умеет выполнить необходимые действия при передаче `--unregister`. Можно производить отмену регистрации серверов явным способом с помощью вызова `UnregisterServer`, имеющегося в конце сценария, но такой способ используется реже.

Вероятно, наиболее интересную часть этого кода представляют особые присвоения *атрибутам класса* в начале класса Python. Эти аннотации класса обеспечивают настройки для регистрации сервера (атрибуты `_reg_`), ограничения доступа (имена `_public_`) и др. Такие атрибуты специфичны для среды COM Python и служат для настройки сервера.

Например, `_reg_class_spec_` просто задает имена модуля и класса Python, разделенные точкой. Если атрибут задан, имеющийся интерпретатор Python в соответствии с его значением импортирует модуль и создает экземпляр класса Python при обращении к нему клиента.[1]

---

[1] Однако заметьте, что в атрибуте `_reg_class_spec_` больше нет строгой необходимости, и если его не указывать, можно избежать ряда проблем с PYTHONPATH. Ввиду того что такие установки подвержены изменениям, необходимо всегда обращаться к справочным материалам в новейших пакетах расширений Windows за подробностями относительно этого и других атрибутов аннотации класса.

С помощью других атрибутов можно идентифицировать сервер в реестре Windows. Атрибут `_reg_clsid_`, например, дает *глобально уникальный идентификатор* (GUID) сервера, который должен быть различным для всех создаваемых COM-серверов. Иными словами, не используйте значение, приведенное в этом сценарии. Вместо этого сделайте то же, что и я, для создания этого ID и вставьте в сценарий результат, полученный на своей машине:[1]

```
A:\>python
>>> import pythoncom
>>> pythoncom.CreateGuid()
<iid:{1BA63CC0-7CF8-11D4-98D8-BB74DD3DDE3C}>
```

GUID генерируются инструментом, поставляемым вместе с пакетом расширения Windows; импортируйте и вызовите функцию `pythoncom.CreateGuid()` и вставьте возвращенный ей текст в свой сценарий. Windows использует ID, проштампованный в сетевой карте машины, чтобы получить сложный ID, который должен оказаться уникальным для всех серверов и машин. Более символическая строка Program ID, `_reg_progid_`, тоже может использоваться клиентами в качестве имен серверов, но вероятность ее уникальности меньше.

В остальном класс сервера состоит из методов на чистом Python, которые реализуют экспортируемое поведение сервера; это то, к чему могут обращаться клиенты. После того как этот сервер Python аннотирован, запрограммирован и зарегистрирован, он может использоваться любым поддерживающим COM языком. Например, программы, написанные на Visual Basic, C++, Delphi и Python, могут обращаться к открытым методам и атрибутам сервера через COM; конечно, другие программы Python могут просто импортировать этот модуль, но цель COM в том, чтобы открыть компоненты для более широкого повторного использования.[2]

Использование сервера Python из клиента Python

Приведем этот COM-сервер Python в действие. Сценарий Python в примере 15.15 тестирует сервер двумя способами: сначала импортируя и вызывая его непосредственно, а затем применяя показанные ранее COM-интерфейсы Python для клиента, чтобы вызвать его более косвенным образом. При прохождении через COM символический ID программы `PythonServers.MyServer`, который мы дали серверу (установив атрибут класса `_reg_progid_`), может быть использован для соединения с этим сервером из любого языка (включая Python).

Пример 15.15. PP2E\Internet\Other\Com\comserver-test.py

```
###############################################################
# Проверка COM-сервера, написанного на Python, из Python двумя способами
###############################################################
```

[1] Приглашение `A:/>` появилось здесь только потому, что я копировал сценарии COM на флоппи, чтобы выполнять на машине, где установлено расширение `win32all`. У вас должен получиться запуск из каталога в дереве примеров, где находятся эти сценарии.

[2] Но следует знать о нескольких правилах работы с типами. В Python 1.5.2 COM-серверы, написанные на Python, должны следить за тем, чтобы использовать фиксированное число аргументов функции и преобразовывать передаваемые строки с помощью встроенной функции `str`. Последнее из этих ограничений возникает потому, что COM передает строки в виде строк Unicode. Но поскольку Python 1.6 и 2.0 теперь поддерживают и строки Unicode, и обычные строки, это ограничение должно вскоре исчезнуть. При использовании COM в качестве клиента (например, когда код вызывает COM) можно передать строку или объект Unicode, и преобразование осуществляется автоматически; при программировании сервера COM (например, когда код вызывается COM) строки всегда передаются как объекты Unicode.

```
def testViaPython():                                    # проверка без com
    from comserver import MyServer                      # имя класса Python
    object = MyServer()                                 # действует, как для любого класса
    print object.Hello()
    print object.Square(8)
    print object.version

def testViaCom():
    from win32com.client import Dispatch                # проверка через com на стороне клиента
    server = Dispatch('PythonServers.MyServer')         # использовать имя из реестра Windows
    print server.Hello()                                # вызов открытых методов
    print server.Square(12)
    print server.version                                # доступ к атрибутам

if __name__ == '__main__':
    testViaPython()                                     # тест модуля, сервера
    testViaCom()                                        # объект com сохраняет состояние
    testViaCom()
```

При правильной настройке и регистрации COM-сервера Python можно связаться с ним через этот тестовый сценарий Python. Ниже мы запускаем файлы сервера и клиента из окна консоли MS-DOS (хотя обычно можно запустить их также щелчками мыши). Первая команда запускает сам файл сервера, чтобы зарегистрировать его в COM; вторая запускает тестовый сценарий, чтобы опробовать сервер как импортированный модуль (`testViaPython`) и как сервер, доступный через COM (`testViaCom`):

```
A:\>python comserver.py
Registered: PythonServers.MyServer

A:\>python comserver-test.py
Hello COM server world [1, 1]
64
1.0
Hello COM server world [2, 1]
144
1.0
Hello COM server world [3, 1]
144
1.0

A:\>python comserver.py --unregister
Unregistered: PythonServers.MyServer
```

Обратите внимание на два числа в конце выходных строк `Hello`: они отражают текущие значения глобальной переменной и атрибута экземпляра сервера. Глобальные переменные в модуле сервера сохраняют состояние, пока загружен модуль сервера; напротив, каждый вызов COM `Dispatch` (и класса Python) создает новый экземпляр класса сервера и, следовательно, новые атрибуты экземпляра. Третья команда отменяет регистрацию сервера в COM в качестве этапа уборки мусора. Интересно, что после отмены регистрации сервера им больше нельзя пользоваться, во всяком случае через COM:

```
A:\>python comserver-test.py
Hello COM server world [1, 1]
64
1.0
Traceback (innermost last):
  File "comserver-test.py", line 21, in ?
    testViaCom()                                        # объект com сохраняет состояние
```

```
  File "comserver-test.py", line 14, in testViaCom
    server = Dispatch('PythonServers.MyServer')                # использовать реестр Windows
  ...more deleted...
pywintypes.com_error: (-2147221005, 'Invalid class string', None, None)
```

Использование сервера Python из клиента VB

Только что приведенный сценарий *comserver-test.py* показывает, как использовать COM-сервер Python из COM-клиента Python. После создания и регистрации COM-сервера Python он становится доступен любому языку, у которого есть интерфейс COM. Скажем, в примере 15.16 приведен образец кода, который нужно написать для доступа к серверу Python из Visual Basic. Клиенты, написанные на других языках (например, Delphi или Visual C++), аналогичны, но синтаксис и вызовы для создания экземпляров могут быть другими.

Пример 15.16. PP2E\Internet\Other\Com\comserver-test.bas

```
Sub runpyserver()
    ' use python server from vb client
    ' alt-f8 in word to start macro editor
    Set server = CreateObject("PythonServers.MyServer")
    hello1 = server.hello()
    square = server.square(32)
    pyattr = server.Version
    hello2 = server.hello()
    sep = Chr(10)
    Result = hello1 & sep & square & sep & pyattr & sep & hello2
    MsgBox Result
End Sub
```

Самое сложное (по крайней мере, для тех, кто так же безыскусен в VB, как автор) заключается в том, чтобы заставить этот код выполниться. Поскольку VB встраивается в продукты Microsoft Office, такие как Word, один из подходов заключается в том, чтобы протестировать код в контексте этих систем. Попробуйте следующее: запустите Word, нажмите одновременно <Alt> и <F8> и вы окажетесь в диалоге макросов Word. Введите в нем имя нового макроса, нажмите Create и вы окажетесь в интерфейсе разработки, где можно вставить показанный код VB и запустить его.

Я не обучаю в этой книге работе с VB, поэтому если у вас не получатся указанные действия, обратитесь к другой документации. Но если наловчиться, все получается довольно просто – выполнение кода VB в этом контексте создает всплывающее окно Word, показанное на рис. 15.8, с результатами обращения VB к нашему COM-серверу Python. Значения глобальной переменной и атрибута экземпляра в конце обоих ответных сообщений `Hello` на этот раз совпадают, потому что мы создаем только один экземпляр класса сервера Python: в VB путем вызова `CreateObject` с ID программы нужного сервера.

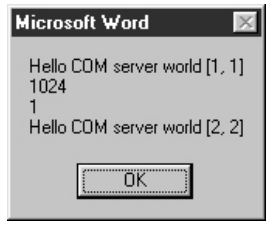


Рис. 15.8. Клиент VB, выполняющий COM-сервер Python

Но поскольку мы теперь умеем встраивать VBScript в страницы HTML, можно запустить код клиента VB другим способом, поместив его на веб-страницу и предоставив его запуск IE. Основу файла HTML в примере 15.17 составляет тот же файл Basic, но код заключен в теги, делающие его настоящей веб-страницей.

Пример 15.17. PP2E\Internet\Other\Com\comserver-test-vbs.html

```
<HTML><BODY>
<P>Run Python COM server from VBScript embedded in HTML via IE</P>
<SCRIPT Language=VBScript>

Sub runpyserver()
    ' use python server from vb client
    ' alt-f8 in word to start macro editor
    Set server = CreateObject("PythonServers.MyServer")
    hello1 = server.hello()
    square = server.square(9)
    pyattr = server.Version
    hello2 = server.hello()
    sep = Chr(10)
    Result = hello1 & sep & square & sep & pyattr & sep & hello2
    MsgBox Result
End Sub

runpyserver()

</SCRIPT>
</BODY></HTML>
```

Здесь очень много маршрутизации, но итоговый результат аналогичен получаемому при самостоятельном выполнении кода VB. Щелчок по этому файлу запускает Internet Explorer (предполагая, что он зарегистрирован как обработчик файлов HTML), который извлекает и запускает встроенный код VBScript, который в свою очередь вызывает COM-сервер Python. То есть IE запускает код VBScript, который запускает код Python – логика управления охватывает три системы, файл HTML, файл Python и реализацию IE. С помощью COM все работает. На рис. 15.9 показан IE в действии, вы-

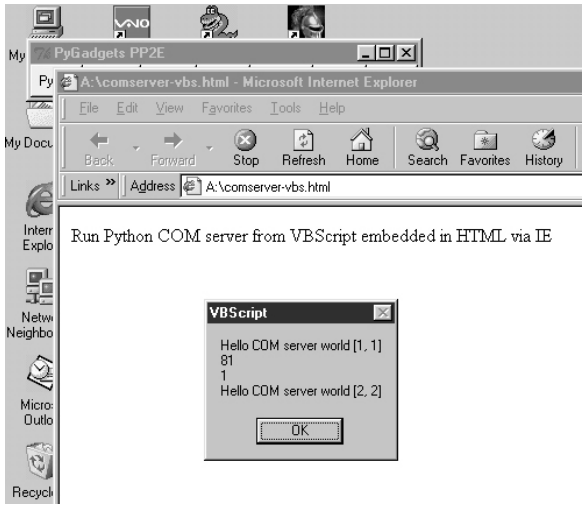


Рис. 15.9. IE, выполняющий клиент VBScript, выполняющий COM-сервер Python

полняющий показанный выше файл HTML. Всплывающее окно генерируется встроенным кодом VB, как и прежде.

Если код вашего клиента выполняется, но генерирует ошибку COM, проверьте, установлен ли пакет `win32all` и находится ли файл модуля сервера в каталоге, входящем в путь поиска Python, а также был ли файл сервера запущен самостоятельно, чтобы зарегистрировать сервер в COM. Если ничто из указанного не помогает, то вы, вероятно, вышли за рамки этой книги. (Ищите детали в дополнительных ресурсах программирования для Windows.)

Общая картина COM

Какое отношение создание COM-серверов на Python имеет к теме Интернета, обсуждаемой в этой главе? Как никак, код Python, встраиваемый в HTML, просто играет роль клиента COM для систем IE или IIS, обычно выполняемых локально. Помимо показа механизма работы таких систем, я остановился здесь на этой теме, поскольку COM, по крайней мере, при широком взгляде на него, тоже занимается сетевыми коммуникациями.

Мы не можем позволить себе в этой книге вникать в детали, но *распределенные* расширения COM делают возможной реализацию COM-серверов, написанных на Python и выполняемых на машинах, сколь угодно удаленных от клиента. Будучи в значительной мере прозрачными для клиентов, вызовы объектов COM, подобные осуществляемым в предыдущих сценариях клиентов, могут предполагать передачу аргументов и результатов по сети. В такой конфигурации COM может использоваться как общая модель реализации архитектуры клиент/сервер и заменить такие технологии, как RPC (Remote Procedure Calls).

Для некоторых приложений этот подход распределенных объектов может даже оказаться жизнеспособной альтернативой другим средствам Python создания сценариев для клиентов и серверов, изучавшимся в этой части книги. Более того, даже в нераспределенном варианте COM является альтернативой технологиям интеграции Python/C на низком уровне, с которыми мы познакомимся в этой книге позже.

Приобретя достаточный опыт, можно с помощью COM легко интегрировать произвольные компоненты и получить стандартный путь написания сценариев и повторного использования систем. Однако COM подразумевает также некоторый уровень издержек, вызываемых непрямой диспетчеризацией, и на момент написания этой книги является решением только для Windows. По этой причине COM обычно характеризуется меньшей быстротой и худшей переносимостью, чем некоторые другие схемы клиент/сервер и интеграции с C, описанные в этой книге. Предпочтительное решение зависит от конкретного приложения.

Как вы, видимо, догадываетесь, о сценариях в Windows можно рассказать значительно больше. Если вас интересуют дополнительные детали, то эта и другие темы, относящиеся к разработке для Windows, превосходно изложены в издании O'Reilly «Python Programming on Win32». Значительная часть работы по написанию сценариев, встроенных в HTML, состоит в применении API открытых объектных моделей, умышленно опущенных в этой книге; дополнительные сведения смотрите в документации Windows.

Python Server Pages

Оставаясь на момент написания этой книги до некоторой степени новинкой, Python Server Pages (PSP) представляет собой технологию для сервера, в которой код JPython встраивается внутрь HTML. PSP является ответом Python на другие подходы к встраиванию сценариев на стороне сервера.

Новый компилятор C# Python

Последние известия: после того, как эта глава была завершена, компания ActiveState (*http://www.activestate.com*) объявила о выпуске нового компилятора для Python. Эта система (с предварительным названием Python.NET) является новой независимой реализацией языка Python (как система Jpython, описанная выше), но компилирует сценарии Python для использования в среде языка Microsoft C# и структуре .NET (системе программных компонентов, основанной на XML и стимулирующей взаимодействие между разными языками). Это открывает дорогу для других ролей и режимов сценариев Python для веб в сфере Windows.

В случае удачи эта новая система компиляции может оказаться третьей реализацией Python (наряду с JPython и стандартной реализацией C) и захватывающим событием для Python в целом. Помимо всего прочего, основанный на C# перенос позволит компилировать сценарии Python в двоичные *.exe*-файлы и разрабатывать их в интегрированной среде Visual Studio. Как и в реализации Jpython, основанной на Java, сценарии кодируются с помощью стандартного базового языка Python, представленного в этой книге, и транслируются для выполнения в базовой системе C#. Кроме того, использование интерфейсов .NET автоматически интегрируется со сценариями Python: классы Python могут создавать подклассы, действовать в качестве и использовать компоненты .NET.

И так же как в случае JPython, эта новая альтернативная реализация Python нацелена на специальную аудиторию и, вероятно, представит наибольший интерес для разработчиков, связанных с интеграцией с C# и .NET. Помимо этого нового компилятора ActiveState также планирует выпустить целый комплект средств разработки для Python; следите за подробностями на сайтах Python и ActiveState.

Механизм сценариев PSP действует способом, сходным с Microsoft Active Server Pages (ASP, описанным ранее) и спецификацией Sun Java Server Pages (JSP). Рискую переполнить чашу терпения к акронимам, но PSP также сравнивается с PHP, языком сценариев для сервера, встраиваемым в HTML. Все эти системы, включая PSP, встраивают сценарии в HTML и выполняют их на сервере для генерации ответного потока, отправляемого в броузер клиента. Для выполнения своих задач сценарии взаимодействуют с API открытой объектной модели. PSP написан на чистом Java и потому переносим на широкий круг платформ (приложения ASP могут работать только на платформах Microsoft).

PSP использует в качестве языка сценариев Jpython – по сообщениям, значительно более подходящий выбор для сценариев сайтов, чем язык Java, используемый в Java Server Pages. При встраивании кода JPython под PSP сценарии получают доступ к большому числу инструментов и расширений Python и JPython. Кроме того, благодаря поддержке в Jpython интеграции с Java сценарии могут обращаться ко всем библиотекам Java.

Мы не имеем возможности подробно обсуждать здесь PSP; беглое представление можно получить из примера 15.18, взятого из документации PSP и иллюстрирующего структуру системы.

Пример 15.18. PP2E\Internet\Other\hello.psp

```
$[
# Генерировать простую страницу сообщения с IP-адресом клиента address
```

```
]$
<HTML><HEAD>
<TITLE>Hello PSP World</TITLE>
</HEAD>
<BODY>
$[include banner.psp]$
<H1>Hello PSP World</H1>
<BR>
$[
Response.write("Hello from PSP, %s." % (Request.server["REMOTE_ADDR"]) )
]$
<BR>
</BODY></HTML>
```

Такого рода страницы устанавливаются на машине сервера с поддержкой PSP, и обращение к ним из броузера производится по URL. В PSP код JPython, встроенный в HTML, заключается между разделителями `$[` и `]$`; все, что находится вне этой пары, просто посылается в броузер клиента, а код между этими маркерами выполняется. Здесь первый блок кода является комментарием JPython (обратите внимание на символ #); второй блок является оператором `include`, который вставляет содержимое другого файла PSP.

Третий участок встроенного кода более полезен. Как и в технологии Active Scripting, встроенный в HTML код Python взаимодействует с контекстом исполнения через API открытых объектов – в данном случае объект `Response` используется для вывода в броузер клиента (аналогично `print` в сценариях CGI), а с помощью `Request` происходит доступ к заголовкам запроса HTTP. В объекте `Request` есть также словарь `params`, который содержит входные параметры `GET` и `POST`, и словарь `cookies`, содержащий данные cookie, сохраняемые у клиента приложением PSP.

Заметьте, что предыдущий пример не труднее реализовать в CGI-сценарии Python с помощью оператора Python `print`, но в больших страницах, содержащих и выполняющих гораздо более сложный код Jpython, выгоды использования PSP становятся более зримыми.

PSP выполняется как сервлет Java и требует, чтобы хост сайта поддерживал Java Servlet API, описание которого находится вне сферы этой книги. Дополнительные сведения о PSP в настоящее время можно получить на сайте *http://www.ciobriefings.com/psp*, однако поищите на *http://www.python.org* другие ссылки, если эта окажется недействующей.

Создание собственных серверов на Python

Большинство модулей Интернета, рассматривавшихся нами в нескольких последних главах, обеспечивает интерфейсы клиента, например FTP и POP, или специальные протоколы сервера, например CGI, которые скрывают сам базовый сервер. Если вы хотите самостоятельно построить на Python сервер, можно сделать это вручную или с помощью инструментов более высокого уровня.

Программные решения

Мы видели в главе 10 «Сетевые сценарии», какой код нужен для построения серверов вручную. Программы Python обычно реализуют серверы, используя чистые вызовы сокетов с потоками, ветвлением или select для параллельной обработки клиентов либо используя модуль `SocketServer`.

В любом случае для обслуживания запросов, осуществляемых в протоколах более высокого уровня, таких как FTP, NNTP и HTTP, нужно слушать порт, соответствующий протоколу, и добавить код для обработки сообщений, допустимых в протоколе. Если пойти этим путем, то модули протоколов для клиентов в стандартной библиотеке Python помогут понять, какие используются соглашения по сообщениям. Возможно, вам также удастся обнаружить примеры протоколов сервера в каталогах Demos и Tools дистрибутива исходного кода Python и в Сети в целом (ищите на *http://www.python.org*). Читайте подробнее в предыдущих главах о написании серверов, основанных на сокетах.

Интерфейсы более высокого уровня представлены в поставляемых вместе с Python готовых реализациях сервера HTTP-протокола веб в виде трех стандартных модулей. `BaseHTTPServer` реализует сам сервер; этот класс является производным от стандартного класса `SocketServer.TCPServer`. `SimpleHTTPServer` и `CGIHTTPServer` реализуют стандартные обработчики входящих запросов HTTP; первый обрабатывает простые запросы файлов веб-страниц, тогда как второй также запускает запрашиваемые сценарии CGI на машине сервера путем ветвления процессов.

Например, для запуска сервера HTTP с поддержкой CGI выполните на машине сервера код Python, подобный приведенному в примере 15.19.

Пример 15.19. PP2E\Internet\Other\webserver.py

```
#!/usr/bin/python
################################################################################
# Реализация на Python сервера HTTP, умеющего запускать сценарии CGI;
# измените корневой каталог соответственно своей машине
################################################################################

import os
from BaseHTTPServer import HTTPServer
from CGIHTTPServer  import CGIHTTPRequestHandler
os.chdir("/home/httpd/html")                        # выполнять в корневом каталоге html
srvraddr = ("", 80)                                 # имя хоста, номер порта
srvrobj  = HTTPServer(srvraddr, CGIHTTPRequestHandler)
srvrobj.serve_forever()                             # выполнять как постоянный демон
```

При этом, конечно, предполагается наличие прав для запуска такого сценария; смотрите в руководстве по библиотеке Python дополнительные сведения о готовом сервере HTTP и модулях обработки запросов. После запуска своего сервера можно обращаться к нему из любого веб-броузера, либо используя модуль Python `httplib`, реализующий клиентскую часть протокола HTTP, либо модуль Python `urllib`, который предоставляет интерфейс типа файла к данным, получаемым из указанного адреса URL (см. примеры `urllib` в главе 11 «Клиентские сценарии» и главе 13 «Более крупные примеры сайтов, часть 1» и используйте URL вида «http:/ /...» для доступа к документам HTTP).

Готовые решения

Наконец, можно установить развитые дружественные к Python серверы и инструменты open source, свободно доступные в Сети. Со временем это тоже может претерпеть изменения, но вот несколько вариантов на текущий момент:

Medusa, asyncore

Система Medusa (*http://www.nightmare.com/medusa*) является архитектурой для построения долговременных высокопроизводительных сетевых серверов на Python и используется в нескольких ответственных системах. Начиная с Python 1.5.2 ядро Medusa входит в стандартную поставку Python в виде модулей библиотеки

`asyncore` и `asynchat`. Эти стандартные модули можно использовать самостоятельно для создания высокопроизводительных сетевых серверов, основанных на асинхронной, мультиплексной однопроцессной модели. В них используется цикл событий с помощью системного вызова `select`, представленного в главе 10 этой книги, чтобы обеспечить одновременность, не порождая потоки или процессы, и они хорошо подходят для обработки недолго живущих транзакций. Смотрите детали в библиотеке Python. Полная система Medusa (не поставляемая с Python) также предоставляет готовые серверы HTTP и FTP; она бесплатна для некоммерческого использования, но в других случаях требует лицензии.

Zope

Если вы делаете что-либо для сервера, подумайте об использовании сервера веб-приложений с открытым исходным кодом Zope, описанного ранее в этой главе и на *http://www.zope.org*. Zope предостовляет полноценную среду веб, реализующую объектную модель, выходящую далеко за рамки стандартных сценариев CGI для серверов. Разработчики Zope создали также развитые серверы (например, Zserver).

Mailman

Если вам нужна поддержка списка рассылки электронной почты, рассмотрите менеджер почтовых списков GNU, известный также под именем Mailman. Он написан на Python и предоставляет надежный, быстрый и богатый функциями инструмент списка рассылки электронной почты для ведения дискуссий. Mailman разрешает пользователям осуществлять подписку через Сеть, поддерживает администрирование через веб, предоставляет шлюзы из электронной почты в телеконференции и встроенную борьбу со спамом (типа «макулатуры» – junk mail). В данный момент узнать подробнее о Mailman можно на *http://www.list.org*.

Apache

Если вы любите риск, вас может заинтересовать веб-сервер с открытым исходным кодом Apache, допускающий высокую степень настройки. Apache – один из доминирующих на сегодняшний день веб-серверов, несмотря на свое бесплатное распространение. Помимо многого другого он поддерживает выполнение сценариев Python для сервера в ряде режимов; подробности о самом Apache смотрите на *http://www.apache.org*.

PyApache

Если вы пользуетесь Apache, поищите на сайте Python информацию о модуле *PyApache* сервера Apache (иногда называемом `mod_pyapache`), который встраивает в Apache интерпретатор Python, чтобы ускорить процесс запуска сценариев Python для сервера. Сценарии CGI непосредственно передаются встроенному интерпретатору, что устраняет расходы на запуск интерпретатора. PyApache также открывает возможность управления внутренними компонентами Apache.

mod_python

Когда я писал эту главу, в мире open source появился еще один пакет для встраивания Python внутрь веб-сервера Apache: `mod_python`, имеющийся на *http://www.modpython.org*. Согласно комментариям к его выходу `mod_python` также позволяет встраивать Python в Apache, существенно увеличивая его производительность и гибкость. Объявление о выходе бета-версии системы появилось в *comp.lang.python* как раз на той неделе, когда был написан этот раздел, поэтому ищите в Сети информацию о его текущем состоянии.

Обязательно следите на *http://www.python.org* за тем, какие события происходят на фронте серверов, а также за новейшими достижениями в технологии создания веб-сценариев на Python в целом.

IV

Разные темы

В этой части книги собраны дополнительные темы приложений Python. Большинство представленных средств может быть использовано в широком круге областей. Здесь вы найдете следующие главы:

- Глава 16 «Базы данных и постоянное хранение». Эта глава освещает часто используемые и развитые технологии Python для хранения информации между запусками программы – файлы DBM, перевод объектов в последовательную форму (сериализация), полки объектов и интерфейсы Python к базам данных SQL.
- Глава 17 «Структуры данных». Здесь рассматриваются технологии Python для реализации более сложных структур данных – стеков, множеств, бинарных деревьев поиска, графов и т. п. В Python они принимают форму реализаций объектов.
- Глава 18 «Текст и язык». В этой главе изучаются инструменты и техника, используемые в Python для синтаксического анализа текстовой информации – разбиение и соединение строк, поиск регулярных выражений, анализ методом рекурсивного спуска и более сложные темы, связанные с языками.

Это последняя часть книги, посвященная чистому Python, и она интенсивно использует инструменты, представленные ранее в книге, особенно библиотеку GUI Tkinter. Например, броузер деревьев используется для иллюстрации различных структур объектов, броузер форм способствует конкретизации понятий баз данных, а GUI калькулятора служит демонстрации понятий обработки языков и повторного использования кода.

16

Базы данных и постоянное хранение

«Дайте мне приказ стоять до конца, но сохранить данные»

До сих пор в этой книге мы использовали Python в системном программировании, разработке GUI и создании сценариев для Интернета – трех самых частых областях применения Python. В следующих трех главах мы бросим беглый взгляд на другие важные темы программирования на Python: постоянное хранение данных, технику работы со структурами данных и средства обработки текста и языков. Ни одна из этих тем не освещается исчерпывающим образом (каждой вполне можно посвятить отдельную книгу), но мы представим примеры работы Python в этих областях и подчеркнем основные идеи. Если какая-либо из этих глав вызовет у вас интерес, то следует обратиться к дополнительным источникам.

Возможности постоянного хранения данных в Python

В этой главе наше внимание сосредоточено на постоянно хранящихся (*persistent*) данных – тех, которые остаются после завершения создавшей их программы. По умолчанию это не так для объектов, создаваемых сценариями: такие вещи, как списки, словари и даже объекты экземпляров классов, располагаются в памяти компьютера и теряются, как только сценарий завершает работу. Чтобы заставить данные жить дольше, требуется сделать что-то особое. В программировании на Python есть по крайней мере пять обычных способов сохранения информации между прогонами программы:

- Плоские файлы: хранение текста и байтов
- Файлы DBM: доступ к строкам по ключу
- Сериализованные объекты: преобразование объектов в потоки байтов
- Файлы полок (shelve): хранение сериализованных объектов в файлах DBM
- Системы баз данных: развитые системы SQL и объектных баз данных

Мы серьезно изучали интерфейсы Python простых (или «плоских») файлов в главе 2 «Системные инструменты» и с того момента пользуемся ими. Python предоставляет стандартный доступ к файловой системе `stdio` (через встроенную функцию `open`), а также на низком уровне через дескрипторы (с помощью встроенного модуля `os`). Для простых задач хранения данных многим сценариям ничего другого не требуется. Чтобы сохранить данные для использования при следующих прогонах программы, нужно записать их в открытый файл на компьютере, а потом считать их обратно из этого файла. Как мы видели, в более сложных задачах Python поддерживает также другие интерфейсы, сходные с файлами, такие как каналы, очереди и сокеты.

Так как мы уже изучали плоские файлы, я не буду больше рассказывать здесь о них. Оставшаяся часть главы знакомит с другими темами из приведенного в начале разде-

ла списка. В конце мы также познакомимся с программой GUI для просмотра содержимого файлов shelve и DBM. Но прежде нам нужно разобраться, что это за звери.

Файлы DBM

Плоские файлы удобно использовать для простых задач постоянного хранения, но обычно они связаны с последовательным режимом обработки. Хотя можно переходить в произвольное место с помощью вызовов `seek`, плоские файлы мало что вносят в структуру данных помимо понятий байтов и текстовых строк.

Файлы DBM, стандартное средство библиотеки Python для управления базами данных, улучшают положение, предоставляя доступ к хранящимся строкам текста по ключу. Они реализуют представление хранящихся данных с произвольным доступом и одним ключом. Например, информация, относящаяся к объектам, может храниться в файле DBM с использованием уникального ключа для каждого объекта и позднее может быть получена непосредственно с помощью того же самого ключа. Файлы DBM реализуются с помощью целого ряда базовых модулей (в том числе одного, написанного на Python), но если у вас есть Python, значит, есть и DBM.

Работа с файлами DBM

Хотя файловые системы DBM должны проделать некоторую работу по отображению соответствия сохраняемых данных ключам для быстрого извлечения (технически для сохранения данных в файлах они обычно используют прием, называемый хешированием), сценариям не приходится беспокоиться о том, что происходит за кулисами. В действительности DBM является одним из простейших способов сохранения информации в Python – файлы DBM ведут себя настолько сходно со словарями, размещаемыми в памяти, что можно забыть о том, что в самом деле вы работаете с файлом. Например, если есть объект файла DBM:

- Индексирование по ключу извлекает данные из файла.
- Присвоение по индексу сохраняет данные в файле.

Объекты файлов DBM поддерживают также стандартные методы словарей, такие как выборка и проверка по списку ключей и удаление по ключу. Сама библиотека DBM скрыта за этой простой моделью. Ввиду простоты перейдем прямо к интерактивному примеру, в котором создается файл DBM и показывается, как работает интерфейс:

```
% python
>>> import anydbm                          # получить интерфейс: dbm, gdbm, ndbm,..
>>> file = anydbm.open('movie', 'c')       # создать файл dbm с именем 'movie'
>>> file['Batman'] = 'Pow!'                # сохранить строку с ключом 'Batman'
>>> file.keys()                            # получить список ключей файла
['Batman']
>>> file['Batman']                         # выбрать значение для ключа 'Batman'
'Pow!'

>>> who  = ['Robin', 'Cat-woman', 'Joker']
>>> what = ['Bang!', 'Splat!', 'Wham!']
>>> for i in range(len(who)):
...     file[who[i]] = what[i]             # добавить еще 3 "записи" "records"
...
>>> file.keys()
['Joker', 'Robin', 'Cat-woman', 'Batman']
>>> len(file), file.has_key('Robin'), file['Joker']
(4, 1, 'Wham!')
>>> file.close()                           # иногда требуется закрытие
```

Внутренне, при импорте anydbm автоматически загружается тот интерфейс DBM, который доступен вашему интерпретатору Python, а при открытии нового файла DBM создается один или более внешних файлов с именами, начинающимися со строки «movie» (о деталях будет сказано чуть ниже). Но после импорта и открытия файл DBM фактически неотличим от словаря. В сущности, объект с именем file можно представить себе как словарь, отображаемый во внешний файл с именем movie.

Однако в отличие от обычных словарей, содержимое file сохраняется в перерыве между прогонами программы Python. Если мы потом вернемся и заново запустим Python, наш словарь будет по-прежнему доступен. Файлы DBM похожи на словари, которые требуется открывать:

```
% python
>>> import anydbm
>>> file = anydbm.open('movie', 'c')          # открыть существующий файл dbm
>>> file['Batman']
'Pow!'

>>> file.keys()                               # keys возвращает список индексов
['Joker', 'Robin', 'Cat-woman', 'Batman']
>>> for key in file.keys(): print key, file[key]
...
Joker Wham!
Robin Bang!
Cat-woman Splat!
Batman Pow!

>>> file['Batman'] = 'Ka-Boom!'               # изменить значение Batman
>>> del file['Robin']                         # удалить элемент Robin
>>> file.close()                              # закрыть после изменений
```

За исключением необходимости импортировать интерфейс и открывать/закрывать файл DBM, программам Python не требуется ничего знать собственно о DBM. Модули DBM добиваются этой интеграции, перегружая операции индексирования и переадресуя их более простым библиотечным средствам. Но глядя на этот код Python, вы никогда бы этого не узнали – файлы DBM выглядят как обычные словари Python, хранящиеся во внешних файлах. Произведенные в них изменения сохраняются неограниченно долго:

```
% python
>>> import anydbm                             # снова открыть файл dbm
>>> file = anydbm.open('movie', 'c')
>>> for key in file.keys(): print key, file[key]
...
Joker Wham!
Cat-woman Splat!
Batman Ka-Boom!
```

Как видите, проще уже некуда. В табл. 16.1 перечислены наиболее частые операции с файлами DBM. Если открыт такой файл, он обрабатывается так, как если бы был словарем Python, находящимся в памяти. Выборка элемента осуществляется индексированием объекта файла по ключу, а запись – путем присвоения по ключу.

Несмотря на интерфейс, похожий на словарь, файлы DBM в действительности отображаются в один или более внешних файлов. Например, когда создается файл GDBM с именем movie, базовый интерфейс gdbm записывает два файла, *movie.dir* и *movie.pag*. Если Python скомпилирован с другим базовым интерфейсом файла с доступом по ключу, на компьютере могут появиться другие внешние файлы.

Таблица 16.1. Операции с файлами DBM

Код Python	Действие	Описание
`import anydbm`	Импорт	Получить `dbm`, `gdbm`... – что установлено
`file = anydbm.open('filename', 'c')`	Открытие[a]	Создать или открыть существующий файл DBM
`file['key'] = 'value'`	Запись	Создать или изменить запись для `key`
`value = file['key']`	Выборка	Загрузить значение записи для `key`
`count = len(file)`	Размер	Возвратить количество хранящихся записей
`index = file.keys()`	Индекс	Получить список имеющихся ключей
`found = file.has_key('key')`	Запрос	Проверить, есть ли запись для `key`
`del file['key']`	Удаление	Удалить запись для `key`
`file.close()`	Закрытие	Закрытие вручную, требуется не всегда

[a] В Python версий 1.5.2 и выше следите за тем, чтобы в качестве второго аргумента при вызове `anydbm.open` передавалась строка `c`, чтобы заставить Python создать файл, если его еще не существует, либо просто открыть его в противном случае. Раньше такой режим действовал по умолчанию. Аргумент `c` не требуется при открытии полок, которые будут обсуждаться далее – они по-прежнему используют по умолчанию режим «открыть или создать», если не указан режим открытия. В `anydbm` можно передавать другие строки режима открытия (например, `n` для создания файла в любом случае и `r` для открытия в режиме только для чтения – новое значение по умолчанию); подробности ищите в справочных материалах по библиотеке.

Технически модуль `anydbm` является интерфейсом к той файловой системе типа DBM, которая имеется в Python. При создании нового файла `anydbm` пытается загрузить модули `dbhash`, `gdbm` и `dbm` интерфейсов файлов с доступом по ключу; при их отсутствии Python автоматически использует универсальную реализацию с именем `dumbdbm`. При открытии существующего файла DBM `anydbm` пытается определить создавшую его систему с помощью модуля `whichdb`. Однако обычно о таких вещах не приходится беспокоиться (если не удалить файлы, которые создает ваша DBM).

Обратите внимание, что файлы DBM иногда требуется явно закрывать, как в последней записи табл. 16.1. Некоторые файлы DBM не требуют вызова закрытия, но другим он нужен, чтобы записать изменения из буфера на диск. В таких системах файл может оказаться поврежден, если не выполнить закрытие. К несчастью, используемая по умолчанию DBM в переносе Python 1.5.2 под Windows, `dbhash` (или иначе `bsddb`), является как раз той системой DBM, которая требует закрывающего вызова во избежание потери данных. Практическое правило состоит в том, чтобы всегда закрывать файлы DBM явно после внесения изменений и перед выходом из программы, чтобы обойти возможные проблемы. Это правило распространяется и на полки – тему, с которой мы познакомимся ниже в этой главе.

Сериализованные объекты

Вероятно, наибольшим ограничением файлов DBM является тип хранимых в них данных: данные, записываемые под ключом, должны быть простой текстовой строкой. Если нужно сохранить объекты Python в файле DBM, иногда можно вручную преобразовывать их в строки и обратно при записи и чтении (например, с помощью

str и eval), но это полностью не решает проблемы. Для объектов Python произвольной сложности, таких как экземпляры классов, требуется нечто другое. Объекты экземпляров классов, к примеру, нельзя впоследствии воссоздать из стандартного строкового представления.

Модуль Python pickle, входящий в стандартную поставку системы Python, обеспечивает требуемое преобразование. Он осуществляет прямое и обратное преобразование находящихся в памяти объектов Python в формат одной линейной строки, пригодной для хранения в плоских файлах, пересылки через сокеты по сети и т. д. Это преобразование объекта в строку часто называют преобразованием в последовательную форму (*serialization*) – произвольные структуры данных, размещенные в памяти, отображаются в последовательном строковом формате. Строковое представление объектов иногда также называют потоком байтов в соответствии с его линейным форматом.

Применение сериализации объектов

Сериализация может показаться сложной, когда сталкиваешься с ней впервые, но отрадно узнать, что Python скрывает все сложности преобразования объектов в строки. На самом деле интерфейсы модуля pickle чрезвычайно просты в употреблении. В следующем перечне приводятся некоторые подробности этого интерфейса.

`P = pickle.Pickler(`*`file`*`)`

Создать новый объект pickler для вывода в последовательном виде в открытый выходной файл file.

`P.dump(`*`object`*`)`

Запись объекта в файл/поток объекта pickler.

`pickle.dump(`*`object, file`*`)`

То же, что два последних вызова вместе: сериализовать объект в открытый файл.

`U = pickle.Unpickler(`*`file`*`)`

Создать объект unpickler, осуществляющий обратное преобразование сериализованной формы из объекта открытого входного файла file.

`object = U.load()`

Читать объект из файла/потока объекта unpickler.

`object = pickle.load(`*`file`*`)`

То же, что два последних вызова вместе: восстановить объект из открытого файла.

`string = pickle.dumps(`*`object`*`)`

Возвратить сериализованное представление object в виде строки символов.

`object = pickle.loads(`*`string`*`)`

Прочесть объект из строки символов вместо файла.

Pickler и Unpickler суть экспортируемые классы. Во всех этих вызовах file является либо объектом открытого файла, либо любым объектом, в котором реализованы те же атрибуты, что и у объектов файла:

- Pickler вызывает метод файла write, передавая в качестве аргумента строку.
- Unpickler вызывает метод файла read со счетчиком байтов и readline без аргументов.

Любой объект, имеющий эти атрибуты, может быть передан в качестве параметра «файл». В частности, file может быть экземпляром класса Python, предоставляющего методы read/write. Благодаря этому можно произвольно отображать сериализованные потоки в объекты, находящиеся в памяти. Можно также пересылать объекты Python

через сеть, заключая сокеты в оболочку, выглядящую как файл для вызовов сериализации у отправителя и вызовов обратного преобразования у получателя (см. детали во врезке «Придание сокетам внешнего вида файлов» в главе 10 «Сетевые сценарии»).

При обычном использовании для сериализации объекта в плоский файл нужно открыть файл в режиме записи и вызвать функцию `dump`; для восстановления объекта нужно снова открыть файл и вызвать `load`:

```
% python
>>> import pickle
>>> table = {'a': [1, 2, 3], 'b': ['spam', 'eggs'], 'c':{'name':'bob'}}
>>> mydb  = open('dbase', 'w')
>>> pickle.dump(table, mydb)

% python
>>> import pickle
>>> mydb  = open('dbase', 'r')
>>> table = pickle.load(mydb)
>>> table

{'b': ['spam', 'eggs'], 'a': [1, 2, 3], 'c': {'name': 'bob'}}
```

Чтобы еще больше упростить этот процесс, модуль примера 16.1 заключает вызовы прямого и обратного преобразований объектов в функции, которые также и открывают файлы, хранящие сериализованную форму объекта.

Пример 16.1. PP2E\Dbase\filepickle.py

```
import pickle

def saveDbase(filename, object):
    file = open(filename, 'w')
    pickle.dump(object, file)        # сериализовать в файл
    file.close()                     # годится любой объект типа файла

def loadDbase(filename):
    file = open(filename, 'r')
    object = pickle.load(file)       # восстановить объект из файла
    file.close()                     # воссоздает объект в памяти
    return object
```

Теперь для сохранения и загрузки просто вызовите функции этого модуля:

```
C:\...\PP2E\Dbase>python
>>> from filepickle import *
>>> L = [0]
>>> D = {'x':0, 'y':L}
>>> table = {'A':L, 'B':D}            # L появляется дважды
>>> saveDbase('myfile', table)        # сериализовать в файл

C:\...\PP2E\Dbase>python
>>> from filepickle import *
>>> table = loadDbase('myfile')       # перегрузить/восстановить
>>> table
{'B': {'x': 0, 'y': [0]}, 'A': [0]}
>>> table['A'][0] = 1
>>> saveDbase('myfile', table)

C:\...\PP2E\Dbase>python
>>> from filepickle import *
>>> print loadDbase('myfile')         # обе L обновились, как ожидалось
{'B': {'x': 0, 'y': [1]}, 'A': [1]}
```

Python может сериализовывать почти все, что угодно, исключая объекты компилированного кода, экземпляры классов, не выполняющие правила импортируемости, с которыми мы встретимся ниже, и экземпляры некоторых встроенных и определяемых пользователем типов, написанных на C или зависящих от преходящих состояний операционной системы (например, объекты открытых файлов не могут быть сериализованы). Если объект не может быть сериализован, возбуждается ошибка `PicklingError`.

За дополнительными сведениями о pickler обратитесь к руководству по библиотеке Python. А листая это руководство (или щелкая по нему), не пропустите описание модуля `cPickle` – реализации `pickle`, написанной на C для повышения быстродействия. Посмотрите также модуль `marshal`, который тоже сериализует объекты, но может обрабатывать только простые типы объектов. Если в вашем Python установлен модуль `shelve`, то он автоматически выбирает для ускорения сериализации `cPickle`, а не `pickle`. Я еще не рассказывал о `shelve`, но сейчас сделаю это.

Файлы shelve

Сериализация позволяет сохранять в файлах и файлоподобных объектах произвольные объекты, но это все же весьма неструктурированный носитель: он не обеспечивает непосредственного простого доступа к членам совокупностей сериализованных объектов. Можно добавлять структуры более высокого уровня, но они не являются внутренне присущими:

- Иногда можно смастерить собственную организацию файлов сериализации более высокого уровня с помощью базовой файловой системы (например, можно записать каждый сериализованный объект в файл, имя которого уникально идентифицирует объект), но такая организация не входит собственно в сериализацию и должна управлятся вручную.
- Можно также записывать словари произвольно большого размера в файл сериализации и обращаться к ним по ключу после загрузки обратно в память, но при этом обратное восстановление из файла загружает весь словарь, а не только тот элемент, в котором мы заинтересованы.

Файлы shelve создают некоторую структуру для совокупностей сериализованных объектов. В файлах этого типа, являющихся стандартной частью системы Python, произвольные объекты Python сохраняются по ключу для извлечения в дальнейшем. В действительности здесь не много нового – файлы shelve служат просто комбинацией файлов DBM и сериализации объектов:

- Чтобы сохранить находящийся в памяти объект по ключу, модуль `shelve` сначала сериализует объект в строку с помощью модуля `pickle`, а затем записывает эту строку в файл DBM по ключу с помощью модуля `anydbm`.
- Чтобы загрузить обратно объект по ключу, модуль `shelve` сначала загружает по ключу строку сериализации объекта из файла DBM с помощью модуля `anydbm`, а затем преобразует ее обратно в исходный объект с помощью модуля `pickle`.

Поскольку `shelve` внутренне использует `pickle`, он может сохранять те же объекты, что и `pickle`: строки, числа, списки, словари, рекурсивные объекты, экземпляры классов и др.

Использование полок

Иными словами, `shelve` служит просто посредником: он сериализует и десериализует объекты, чтобы можно было поместить их в файлы DBM. В конечном итоге полки поз-

воляют записывать почти произвольные объекты Python в файлы по ключу и позднее загружать их обратно по тому же ключу. Однако ваши сценарии никогда не видят всех этих взаимодействий. Подобно файлам DBM, полки предоставляют интерфейс, который выглядит как словарь, который нужно открыть. Чтобы получить доступ к полке, импортируйте модуль и откройте свой файл:

```
import shelve
dbase = shelve.open("mydbase")
```

Внутренне Python открывает файл DBM с именем *mydbase* или создает его, если он еще не существует. Присвоение полке по ключу сохраняет объект:

```
dbase['key'] = object
```

Внутренне это присвоение преобразует объект в сериализованный поток байтов и записывает его по ключу в файл DBM. Обращение к полке по индексу ключа загружает сохраненный объект:

```
value = dbase['key']
```

Внутренне эта операция обращения по индексу загружает строку по ключу из файла DBM и разворачивает ее в объект в памяти, совпадающий с исходным сохраненным. Большинство операций со словарями здесь также поддерживается:

```
len(dbase)      # количество хранящихся элементов
dbase.keys()    # указатель ключей хранящихся элементов
```

И, за исключением некоторых тонких моментов, это все, что касается использования shelve. Полки обрабатываются с использованием обычного синтаксиса словарей Python, поэтому не нужно изучать новый API базы данных. Более того, объекты, сохраняемые на полках и восстанавливаемые с них, являются обычными объектами Python: для сохранения не требуется, чтобы они были экземплярами особых классов или типов. То есть система постоянного хранения Python является внешней по отношению к самим сохраняемым объектам. В табл. 16.2 сведены вместе эти и другие часто используемые операции с полками.

Таблица 16.2. Операции с файлами полок

Код Python	Действие	Описание
`import shelve`	Импорт	Получить `dbm`, `gdbm`... – что установлено
`file = shelve.open('filename')`	Открытие	Создать или открыть существующий файл DBM
`file['key'] = anyvalue`	Запись	Создать или изменить запись для `key`
`value = file['key']`	Выборка	Загрузить значение записи для `key`
`count = len(file)`	Размер	Возвратить количество хранящихся записей
`index = file.keys()`	Индекс	Получить список имеющихся ключей
`found = file. has_key('key')`	Запрос	Проверить, есть ли запись для `key`
`del file['key']`	Удаление	Удалить запись для `key`
`file.close()`	Закрытие	Закрытие вручную, требуется не всегда

Так как полки тоже экспортируют интерфейс, подобный словарю, эта таблица почти идентична таблице операций DBM. Однако здесь имя модуля `anydbm` заменяется на `shelve`, вызовы `open` не требуют второго аргумента `c`, а сохраняемые значения могут

быть практически произвольными типами объектов, а не просто строками. Для надежности все же следует явно закрывать полки после проведенных изменений: полки внутренне используют anydbm, а некоторые базовые DBM требуют выполнить закрытие, чтобы избежать потери или повреждения данных.

Сохранение объектов встроенных типов

Запустим интерактивный сеанс и поэкспериментируем с интерфейсами полок:

```
% python
>>> import shelve
>>> dbase = shelve.open("mydbase")
>>> object1 = ['The', 'bright', ('side', 'of'), ['life']]
>>> object2 = {'name': 'Brian', 'age': 33, 'motto': object1}
>>> dbase['brian']  = object2
>>> dbase['knight'] = {'name': 'Knight', 'motto': 'Ni!'}
>>> dbase.close()
```

Здесь мы открываем полку и сохраняем две довольно сложных структуры данных словаря и списка, просто присваивая их ключам полки. Поскольку shelve внутренне использует pickle, здесь можно использовать почти все – деревья вложенных объектов автоматически сериализуются в строки для хранения. Чтобы загрузить их обратно, просто откройте снова полку и индекс:

```
% python
>>> import shelve
>>> dbase = shelve.open("mydbase")
>>> len(dbase)                      # записей
2

>>> dbase.keys()                    # индекс
['knight', 'brian']

>>> dbase['knight']                 # выборка
{'motto': 'Ni!', 'name': 'Knight'}

>>> for row in dbase.keys():
...     print row, '=>'
...     for field in dbase[row].keys():
...         print '  ', field, '=', dbase[row][field]
...
knight =>
   motto = Ni!
   name = Knight
brian =>
   motto = ['The', 'bright', ('side', 'of'), ['life']]
   age = 33
   name = Brian
```

Вложенные циклы в конце этого сеанса проходят вложенные словари – внешний просматривает полку, а внутренний просматривает объекты, хранящиеся на полке. Важно отметить, что для записи и выборки этих постоянных объектов, так же как для их обработки после загрузки, используется обычный синтаксис Python.

Хранение экземпляров классов

Более полезным типом объектов, которые можно хранить на полке, являются экземпляры классов. Поскольку в их атрибутах записывается состояние, а унаследованные

методы определяют поведение, постоянные объекты классов, в сущности, выполняют роль как записей базы данных, так и программ обработки баз данных. Рассмотрим простой класс, показанный в примере 16.2, с помощью которого моделируются люди.

Пример 16.2. PP2E\Dbase\person.py (version 1)

```
# объект человека: поля + поведение

class Person:
    def __init__(self, name, job, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay                 # действительные данные экземпляра
    def tax(self):
        return self.pay * 0.25          # вычисляется при вызове
    def info(self):
        return self.name, self.job, self.pay, self.tax()
```

Можно создавать из этого класса постоянные объекты, просто создавая экземпляры как обычно и сохраняя их по ключу на открытой полке:

```
C:\...\PP2E\Dbase>python
>>> from person import Person
>>> bob   = Person('bob', 'psychologist', 70000)
>>> emily = Person('emily', 'teacher', 40000)
>>>
>>> import shelve
>>> dbase = shelve.open('cast')          # создать новую полку
>>> for obj in (bob, emily):             # сохранить объекты
>>>     dbase[obj.name] = obj            # использовать имя как ключ
>>> dbase.close()                        # требуется для bsddb
```

Когда позднее мы снова загрузим эти объекты в сеансе или сценарии Python, они воссоздадутся в памяти в том виде, в каком находились в момент сохранения:

```
C:\...\PP2E\Dbase>python
>>> import shelve
>>> dbase = shelve.open('cast')          # заново открыть полку
>>>
>>> dbase.keys()                         # здесь оба объекта
['emily', 'bob']
>>> print dbase['emily']
<person.Person instance at 799940>
>>>
>>> print dbase['bob'].tax()             # вызов: налог для bob
17500.0
```

Обратите внимание, что вызов метода `tax` для объекта `bob` работает несмотря на то, что мы не импортировали класс `Person`. Python достаточно сообразителен, чтобы снова связать объект с исходным классом после восстановления из сериализованной формы и сделать все исходные методы доступными через загруженные объекты.

Изменение классов хранимых объектов

Технически Python повторно импортирует класс для воссоздания его сохраненных экземпляров при их загрузке и восстановлении. Вот как это действует:

Запись

Когда Python сериализует экземпляр класса, чтобы сохранить его на полке, он сохраняет атрибуты и ссылку на класс экземпляра. В действительности Python се-

риализует и записывает словарь атрибутов экземпляра `__dict__` вместе с информацией об исходном файле модуля класса.

Выборка

Когда Python восстанавливает экземпляр класса, взятый с полки, он воссоздает в памяти объект экземпляра, повторно импортируя класс и присваивая сохраненный словарь атрибутов новому пустому экземпляру класса.

Главное в этом то, что сам класс не хранится вместе со своими экземплярами, а заново импортируется позднее, когда загружаются экземпляры. В результате, модифицировав внешние классы в файлах модулей, можно изменить способ, которым интерпретируются и используются данные сохраненных объектов, не изменяя сами эти хранящиеся объекты. Это похоже на то, как если бы класс был программой, обрабатывающей хранящиеся записи.

Чтобы проиллюстрировать это, предположим, что класс `Person` предыдущего раздела был изменен, как показано в исходном коде примера 16.3.

Пример 16.3. PP2E\Dbase\person.py (version 2)

```
# объект человека: поля + поведение
# изменение: метод tax стал вычисляемым атрибутом

class Person:
    def __init__(self, name, job, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay               # действительные данные экземпляра
    def __getattr__(self, attr):      # при person.attr
        if attr == 'tax':
            return self.pay * 0.30    # вычисляется при доступе
        else:
            raise AttributeError      # другие неизвестные имена
    def info(self):
        return self.name, self.job, self.pay, self.tax
```

В этой версии устанавливается новая ставка налога (30%), вводится метод `__getattr__` для перегрузки доступа к атрибуту класса и убран исходный метод `tax`. Ссылки на атрибут Tax перехватываются и вычисляются:

```
C:\...\PP2E\Dbase>python
>>> import shelve
>>> dbase = shelve.open('cast')       # заново открыть полку
>>>
>>> print dbase.keys()                # здесь оба объекта
['emily', 'bob']
>>> print dbase['emily']
<person.Person instance at 79aea0>
>>>
>>> print dbase['bob'].tax            # ненужно вызывать tax()
21000.0
```

Класс изменился, так что к `tax` теперь можно просто обратиться как к атрибуту, а не вызывать его как метод. Кроме того, поскольку ставка налога в классе изменена, Бобу придется на этот раз платить больше. Конечно, этот пример искусственный, но при правильном использовании такое разделение классов и постоянных экземпляров может исключить использование многих обычных программ обновления баз данных – в большинстве случаев можно просто изменить класс, а не каждый хранящийся экземпляр, чтобы добиться нового поведения.

Ограничения shelve

Хотя обычно пользоваться полками просто, есть несколько шероховатостей, о которых следует помнить.

Ключи должны быть строками

Во-первых, хотя сохранять можно произвольные объекты, ключи все же должны быть строками. Следующий код не будет выполнен, если сначала вручную не преобразовать целое число 42 в строку «42»:

```
dbase[42] = value      # отказ, но str(42) работает
```

Это является отличием от словарей, размещаемых в памяти, которые допускают использование в качестве ключей любых постоянных объектов, и проистекает из внутреннего использования полками файлов DBM.

Объекты уникальны только по ключу

Хотя модуль `shelve` достаточно сообразителен, чтобы обнаруживать множественные случаи вложенного объекта и воссоздавать только один экземпляр при загрузке, это относится только к данной позиции:

```
dbase[key] = [object, object]    # OK: только один экземпляр сохраняется и извлекается

dbase[key1] = object
dbase[key2] = object             # плохо?: два экземпляра объекта на полке
```

При выборке `key1` и `key2` они указывают на независимые копии первоначального общего объекта; если этот объект изменяем, изменения в одном из них не будут отражены в другом. В действительности это проистекает из того обстоятельства, что каждое присвоение ключу запускает независимую операцию сериализации – pickler обнаруживает повторяющиеся объекты, но только в рамках одного обращения к pickle. Это может не коснуться вас на практике и преодолевается введением дополнительной логики, но объект может дублироваться, если он соответствует нескольким ключам.

Обновления должны работать с полками в режиме «загрузить-модифицировать-сохранить»

Так как объекты, загруженные с полки, не знают, что они пришли с полки, то операции, изменяющие части загруженного объекта, касаются только экземпляра, находящегося в памяти, а не данных на полке:

```
dbase[key].attr = value   # полка не изменилась
```

Чтобы действительно изменить объект, хранящийся на полке, нужно загрузить его в память, изменить его части и записать обратно на полку целиком присвоением по ключу:

```
object = dbase[key]       # загрузить
object.attr = value       # модифицировать
dbase[key] = object       # записать обратно – полка изменилась
```

Одновременное обновление не допускается

Как мы узнали в конце главы 14 «Более крупные примеры сайтов, часть 2», модуль `shelve` в настоящее время не поддерживает одновременные обновления. Одновременное чтение допускается, но записывающий процесс должен получить исключительный доступ к полке. Полку можно разрушить, если несколько процессов будут запи-

сывать в нее одновременно, а это может часто происходить в таких областях, как сценарии CGI для серверов. Если доступ к полкам может потребоваться нескольким процессам, сделайте для обновлений оболочку с вызовом встроенной функции `fcntl.flock`, как об этом говорилось в главе 14.

Ограничения класса Pickler

Помимо указанных ограничений полок сохранение экземпляров классов на полках вводит ряд правил, о которых необходимо знать. В действительности они налагаются модулем `pickle`, а не `shelve`, поэтому следуйте им и тогда, когда сохраняете объекты класса непосредственно с помощью `pickle`.

Классы должны быть импортируемыми

Объект Python pickler при сериализации объекта экземпляра сохраняет только атрибуты экземпляра и затем заново импортирует класс, чтобы воссоздать экземпляр. По этой причине, когда объекты восстанавливаются из сериализованного вида, должна быть возможность импорта классов сохраненных объектов – они должны программироваться как невложенные на верхнем уровне файла модуля, который можно найти по PYTHONPATH. Далее, когда сериализуются экземпляры, они должны ассоциироваться с действительным модулем, а не со сценарием верхнего уровня (с именем модуля `__main__`), и нужно следить за тем, чтобы модули классов не перемещались после сохранения экземпляров. При восстановлении экземпляра Python должен найти модуль своего класса в пути PYTHONPATH по имени исходного модуля (включая префиксы пути пакета) и загрузить класс из этого модуля, используя первоначальное имя класса. Если модуль или класс перемещены или переименованы, класс нельзя будет найти.

Изменения в классе должны обеспечивать обратную совместимость

Хотя Python позволяет изменить класс в то время, когда его экземпляры хранятся на полке, эти изменения должны быть обратно совместимыми с уже сохраненными объектами. Например, нельзя изменять класс так, чтобы он ждал атрибута, отсутствующего в уже хранящихся постоянных экземплярах, если только не изменить предварительно эти хранящиеся экземпляры или предоставить дополнительные протоколы преобразования для класса.

В предыдущей версии Python классы постоянных объектов также должны были иметь конструкторы без аргументов или обеспечивать значения по умолчанию для всех аргументов конструктора (подобно понятию конструктора copy в C++). Это ограничение снято в Python 1.5.2 – классы без значений по умолчанию в аргументах конструктора нормально работают при сериализации.[1]

---

[1] Тонкое место: внутренне Python теперь не вызывает класс для воссоздания сериализованного экземпляра, а вместо этого создает общий объект класса, вставляет атрибуты экземпляра и прямо устанавливает указатель экземпляра `__class__`, чтобы тот указывал на исходный класс. При этом не требуются значения по умолчанию, но это также означает, что конструкторы класса `__init__` больше не вызываются при восстановлении объектов, если не пользоваться дополнительными методами для принудительного вызова. Смотрите дополнительные подробности в руководстве по библиотеке, а также исходный код модуля `pickle` (*pickle.py* в библиотеке исходного кода), если вас интересует, как это работает. Еще лучше, посмотрите модуль `formtable`, приведенный далее в этой главе, – он делает нечто очень сходное со ссылками `__class__`, чтобы создать объект экземпляра из класса и словаря атрибутов, не вызывая конструктор класса `__init__`. В результате в классах для записей, которые просматривает PyForm, значения аргументов конструктора по умолчанию становятся необязательными, но идея та же самая.

Другие ограничения постоянного хранения

Запомните, что помимо указанных ограничений файлы, создаваемые базовой системой DBM, могут оказаться совместимыми не со всеми существующими реализациями DBM. Например, файл, созданный `gdbm`, может не читаться в Python, у которого установлен другой модуль DBM, если только явно не импортировать `gdbm` вместо `anydbm` (если он вообще установлен). Если переносимость файлов DBM имеет значение, обеспечьте использование всеми Python, которые будут читать ваши данные, совместимых модулей DBM.

Наконец, хотя полки постоянно хранят объекты, они не являются в действительности системами объектно-ориентированных баз данных (OODB). В таких системах реализуются также такие функции, как декомпозиция объектов и отложенная («lazy») выборка компонентов, основанные на генерируемых ID объектов: части больших объектов загружаются в память только во время доступа к ним. Можно расширить полки, чтобы они поддерживали такие функции, но вам не нужно этого делать – система Zope, описанная в главе 15 «Более сложные темы Интернета», содержит реализацию более полной системы OODB. Она построена поверх встроенной поддержки постоянного хранения в Python, но предлагает дополнительные функции для более развитых хранилищ данных. Информация и ссылки есть в предыдущей главе.

Интерфейсы баз данных SQL

Полки являются мощным инструментом: они дают возможность сценариям подбрасывать объекты Python в файлы с доступом по ключу и затем загружать их обратно за один шаг. Тем не менее они не являются полновесной системой баз данных; доступ к объектам (записям) происходит по единственному ключу, и нет понятия запросов SQL. Полки представляют собой как бы базу данных с единственным индексом и отсутствием поддержки обработки других запросов.

Можно построить интерфейс с множественными индексами, чтобы хранить данные на нескольких полках, но это нетривиальная задача, требующая написания расширений вручную (см. прототип этой идеи в модуле `dbaseindexed` системы PyErrata в конце главы 14).

Для удовлетворения потребностей хранения промышленного масштаба в Python есть поддержка реляционных СУБД. В настоящее время имеются бесплатные интерфейсы, позволяющие сценариям Python использовать все распространенные системы баз данных – как свободно распространяемые, так и коммерческие: Oracle, Sybase, Informix, mSql, MySql, Interbase, Postgres, ODBC и др. Кроме того, сообщество Python определило спецификацию API баз данных, переносимым образом работающего с рядом пакетов баз данных. Сценарии, написанные для этого API, могут переходить на пакеты других поставщиков баз данных с минимальными изменениями в исходном коде или вообще без них.

Обзор интерфейса

В отличие от всех тем, относящихся к постоянному хранению и представленных к настоящему моменту в этой главе и в этой книге, базы данных SQL являются необязательными расширениями, не являющимися частью самого Python, и для понимания их интерфейсов требуется знание SQL. Поскольку в этой книге нет места для обучения SQL, в данном разделе приводится краткий обзор API; за дополнительными сведениями обратитесь к справочникам по SQL и ресурсам API баз данных, указанным в следующем разделе.

Хорошая весть состоит в том, что можно обращаться к базам данных SQL из Python с помощью простой и переносимой модели. Спецификация API баз данных Python определяет интерфейс для связи с используемыми системами баз данных из сценариев Python. Интерфейсы баз данных конкретных поставщиков для Python могут не вполне соответствовать этому API, но все расширения баз данных для Python имеют незначительные отклонения. В Python базы данных SQL основываются на нескольких понятиях:

- *Объекты соединений* представляют соединение с базой данных, служат интерфейсом для операций отката и фиксации и создают объекты курсоров.
- *Объекты курсоров* представляют одну команду SQL, посылаемую в виде строки, и могут использоватся для прохода по результатам, возвращаемым командой SQL.
- *Результаты запроса* команды SQL `select` возвращаются в сценарии в виде списков кортежей Python, представляющих таблицы строк баз данных. Внутри этих кортежей строк значения полей являются обычными объектами Python, такими как строки, целые и числа с плавающей точкой, либо специальными типами (например, `[('bob',38), ('emily',37)]`).

Помимо этого, API определяет стандартный набор типов исключительных ситуаций баз данных, конструкторы специальных типов баз данных (например, null и дат) и информационные вызовы.

Например, чтобы установить соединение с базой данных через совместимый с Python API интерфейс для Oracle, который можно получить у Digital Creations, установите расширение, а затем выполните строку такого вида:

```
connobj = Connect("user/password@system")
```

Содержимое строкового аргумента зависит от базы данных и производителя, но обычно это данные, с помощью которых производится регистрация в системе базы данных. Получив объект соединения, можно делать с ним разные вещи, в том числе:

```
connobj.close()              закрыть соединение сейчас (не при __del__ объекта)
connobj.commit()             зафиксировать текущие транзакции в базе данных
connobj.rollback()           откатить базу данных в начало текущих транзакций
connobj.getSource(proc)      загрузить код хранимой процедуры
```

Одно из наиболее полезных действий с объектом соединения заключается в создании объекта курсора:

```
cursobj = connobj.cursor()   возвратить новый объект курсора для выполнения SQL
```

У объектов курсоров тоже есть набор методов (например, `close`, позволяющий закрыть курсор раньше, чем будет выполнен его деструктор), самым важным из которых является, видимо, этот:

```
cursobj.execute(sqlstring [, parm, parm,...])  выполнить запрос SQL или командную строку
```

С помощью метода `execute` можно выполнять различные строки предложений SQL:

- команды определения *DDL* (например, CREATE TABLE)
- команды модификации *DML* (например, UPDATE или INSERT)
- команды запросов *DQL* (например, SELECT)

Для команд DML `execute` возвращает количество измененных строк. Для команд запросов DQL возвращается `None`, и для завершения операции нужно вызвать один из методов `fetch`:

```
tuple      = cursobj.fetchone()      получить следующую строку из результата запроса
```

```
listoftuple = cursobj.fetchmany([size])    получить следующую группу строк
                                           из результата запроса
listoftuple = cursobj.fetchall()           получить все оставшиеся строки
                                           из результата запроса
```

После получения результатов метода fetch данные таблицы обрабатываются с помощью обычных операций с объектами списков и кортежей Python (например, можно обойти кортежи в списке результатов `fetchall` с помощью простого цикла `for`). Большинство интерфейсов баз данных Python позволяет также задавать значения, которые должны быть переданы в строки команд SQL, задавая цель и кортеж параметров. Например:

```
query = 'SELECT name, shoesize FROM spam WHERE job = ? AND age = ?'
cursobj.execute(query, (value1, value2))
results = cursobj.fetchall()
for row in results: ...
```

В данном случае интерфейс базы данных использует подготовленные команды (для оптимизации и удобства) и правильно передает параметры в базу данных независимо от их типов в Python. Обозначения параметров в интерфейсах некоторых баз данных могут быть иными (например, «:p1» и «:p2» вместо «?» и «?»); во всяком случае это не то же самое, что оператор Python форматирования строки «%».

Наконец, если база данных поддерживает хранимые процедуры, обычно можно вызывать их с помощью метода `callproc` или передав методу `execute` строку команды SQL CALL или EXEC; для извлечения результатов служит одна из разновидностей метода `fetch`.

Ресурсы

Об интерфейсах баз данных можно рассказать больше, чем это сделано здесь, но дополнительную документацию по API легко получить из Интернета. Наилучшим ресурсом информации по расширениям баз данных на сегодняшний день является, вероятно, домашняя страница группы интересов Python (SIG) по базам данных. Зайдите на *http://www.python.org*, щелкните по ссылке SIGs вверху и перейдите на страницу группы по базам данных (или прямо отправьтесь на *http://www.python.org/sigs/db-sig*, текущий адрес страницы на момент написания). Там вы найдете документацию по API, ссылки на модули расширений баз данных конкретных поставщиков и др.

Находясь на *python.org*, изучите также пакет базы данных Gadfly – основанное на SQL расширение баз данных специально для Python, обладающее широкой переносимостью, возможностью соединений через сокеты для режимов клиент/сервер и др. Gadfly загружает данные в память, поэтому в настоящее время область его применения несколько ограничена. С другой стороны, он идеален для создания прототипов приложений баз данных – можете подождать с выпиской чека продавцу, пока не наступит момент для масштабирования системы перед развертыванием. Кроме того, Gadfly и сам по себе годится для многих применений – не каждой системе требуется большое хранилище данных, но выиграть от мощи SQL могут многие.

PyForm: средство просмотра постоянных объектов

Вместо того чтобы заниматься деталями интерфейса баз данных, которые можно легко найти на *python.org*, я завершу эту главу показом того, как можно соединить технологию GUI, с которой мы познакомились ранее в книге, с технологиями постоянного хранения, представленными в этой главе. Этот раздел представляет *PyForm*, осно-

ванный на Tkinter GUI, предназначенный для просмотра и редактирования таблиц записей:

- *Таблицы*, которые просматривает программа, могут быть полками, файлами DBM, словарями, находящимися в памяти, или любым другим объектом, который имеет вид словаря.
- *Записи* в просматриваемых таблицах могут быть экземплярами классов, простыми словарями, строками или любыми другими объектами, которые могут транслироваться в словарь и обратно.

Хотя этот пример демонстрирует GUI и постоянное хранение, он также иллюстрирует технологию проектирования Python. Чтобы сделать реализацию простой и не зависящей от типов, GUI PyForm написан в предположении, что таблицы имеют вид *словаря словарей*. Для поддержки различных типов таблиц и записей PyForm полагается на отдельные классы-оболочки, транслирующие таблицы и записи в предполагаемый протокол:

- На верхнем уровне таблицы трансляция происходит просто – полки, файлы DBM и словари в оперативной памяти располагают одинаковым основанным на ключе интерфейсом.
- Для вложенного уровня записей в коде GUI предполагается, что хранимые элементы тоже имеют интерфейс типа словаря, но операции со словарями перехватываются классами, чтобы сделать записи совместимыми с протоколом PyForm. Записи, хранящиеся в виде строк, преобразуются в объекты словарей и обратно при выборке и сохранении; записи, хранящиеся в виде экземпляров классов, транслируются в словари атрибутов и обратно. Более специальные виды трансляции можно добавить в новых классах-оболочках таблиц.

В результате PyForm можно использовать для просмотра и редактирования большого числа разных типов таблиц, несмотря на предполагаемый интерфейс словаря. При просмотре с помощью PyForm полок и файлов DBM изменения в таблицах, производимые в GUI, оказываются постоянными – они сохраняются в базовых файлах. При просмотре полки с экземплярами классов PyForm становится, по существу, GUI клиента для простой объектной базы данных, если она создана с помощью стандартных средств Python, обеспечивающих постоянное хранение.

Делаем трудным способом

Прежде чем заняться GUI, разберемся, однако, зачем он нужен. Для экспериментов с полками в целом я сначала подготовил тестовый файл данных. Сценарий примера 16.4 кодирует словарь, используемый для заполнения баз данных (`cast`), а также класс, используемый для заполнения полок экземпляров классов (`Actor`).

Пример 16.4. PP2E\Dbase\testdata.py

```
# определения для тестирования полок, dbm и formgui

cast = {
    'rob':   {'name': ('Rob', 'P'),   'job': 'writer', 'spouse': 'Laura'},
    'buddy': {'name': ('Buddy', 'S'), 'job': 'writer', 'spouse': 'Pickles'},
    'sally': {'name': ('Sally', 'R'), 'job': 'writer'},
    'laura': {'name': ('Laura', 'P'), 'spouse': 'Rob',    'kids':1},
    'milly': {'name': ('Milly', '?'), 'spouse': 'Jerry',  'kids':2},
    'mel':   {'name': ('Mel', 'C'),   'job': 'producer'},
    'alan':  {'name': ('Alan', 'B'),  'job': 'comedian'}
}
```

```
class Actor:                                    # класс уровня невложенного файла
    def __init__(self, name=(), job=''):       # не нужны значения аргументов по умолчанию,
        self.name = name                        # для нового pickler или formgui
        self.job  = job
    def __setattr__(self, attr, value):        # по setattr(): проверить
        if attr == 'kids' and value > 10:       # но все равно установить
            print 'validation error: kids =', value
        if attr == 'name' and type(value) != type(()):
            print 'validation error: name type =', type(value)
        self.__dict__[attr] = value             # не запускать __setattr__
```

Здесь объект `cast` предназначен для представления таблицы записей (в действительности это словарь словарей, когда записывается в таком синтаксисе Python). Теперь при наличии этих тестовых данных легко заполнить полку словарями `cast`. Нужно просто открыть полку и копировать `cast`, один ключ за другим, как в примере 16.5.

Пример 16.5. PP2E\Dbase\castinit.py

```
import shelve
from testdata import cast
db = shelve.open('data/castfile')      # создать новую полку
for key in cast.keys():
    db[key] = cast[key]                # сохранить словари на полке
```

После этого почти так же просто проверить результат с помощью сценария, выводящего содержимое полки, как показано в примере 16.6.

Пример 16.6. PP2E\Dbase\castdump.py

```
import shelve
db = shelve.open('data/castfile')      # повторно открыть полку
for key in db.keys():                  # показать все ключи, значения
    print key, db[key]
```

Вот эти два сценария в действии, заполняющие и отображающие полку словарей:

```
C:\...\PP2E\Dbase>python castinit.py
C:\...\PP2E\Dbase>python castdump.py
alan {'job': 'comedian', 'name': ('Alan', 'B')}
mel {'job': 'producer', 'name': ('Mel', 'C')}
buddy {'spouse': 'Pickles', 'job': 'writer', 'name': ('Buddy', 'S')}
sally {'job': 'writer', 'name': ('Sally', 'R')}
rob {'spouse': 'Laura', 'job': 'writer', 'name': ('Rob', 'P')}
milly {'spouse': 'Jerry', 'name': ('Milly', '?'), 'kids': 2}
laura {'spouse': 'Rob', 'name': ('Laura', 'P'), 'kids': 1}
```

Пока все хорошо. Но здесь мы приближаемся к пределу ручной обработки полок: чтобы модифицировать полку, требуются значительно более общие инструменты. Можно написать маленькие сценарии Python, каждый из которых выполняет весьма специфические обновления. Можно даже временно обойтись вводом вручную таких команд в интерактивном интерпретаторе:

```
>>> import shelve
>>> db  = shelve.open('data/castfile')
>>> rec = db['rob']
>>> rec['job'] = 'hacker'
>>> db['rob'] = rec
```

С любыми базами данных, кроме наиболее тривиальных, это быстро наскучит – особенно для конечных пользователей системы. Что действительно могло бы понравить-

ся, так это GUI, позволяющий просматривать и редактировать полки произвольным образом и легко запускаемый из других программ и сценариев, как в примере 16.7.

Пример 16.7. PP2E\Dbase\castview.py

```
import shelve
from TableBrowser.formgui import FormGui     # после initcast
db = shelve.open('data/castfile')            # снова открыть файл полки
FormGui(db).mainloop()                       # просмотр имеющихся полок словарей
```

Как заставить этот конкретный сценарий работать, станет ясно из следующего раздела.

Делаем в графическом режиме

Путь, прослеживаемый в последнем разделе, это то, что привело меня к написанию PyForm, инструмента GUI для редактирования произвольных таблиц записей. Когда таблицами служат полки и файлы DBM, данные, отображаемые PyForm, постоянны: они продолжают жить после исчезновения GUI. Поэтому можно рассматривать PyForm как броузер простых баз данных.

Код GUI PyForm

Все интерфейсы GUI, используемые PyForm, мы уже встречали ранее в этой книге, поэтому я не стану здесь подробно обсуждать во все детали их реализации (основу можно найти в главах части II «Программирование GUI»). Но прежде чем изучать код, посмотрим, что он делает. Рис. 16.1 демонстрирует работу PyForm под Windows при просмотре полки с постоянными объектами экземпляров, созданных из класса `Actor` модуля `testdata`. Этот код работает таким же образом под Linux и Macs, хотя внешний вид несколько иной.

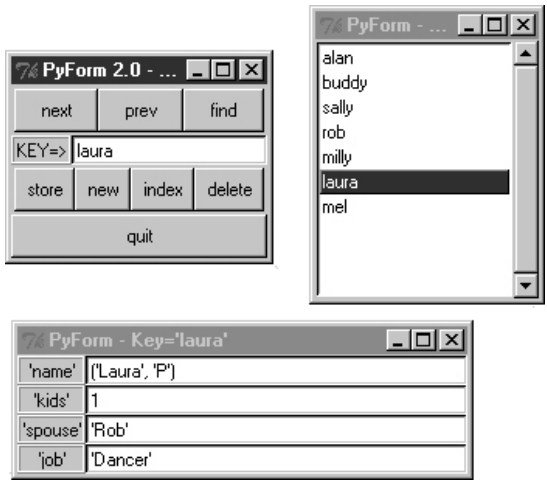


Рис. 16.1. PyForm выводит полку с объектами Actor

Интерфейс PyForm к просматриваемой таблице состоит из трех окон; все окна упакованы для правильного расширения и обрезки, как установлено правилами, изучавшимися нами ранее в этой книге. Окно в левом верхнем углу рис. 16.1 является главным и создается при запуске PyForm; в нем есть кнопки для навигации по таблице, поиска элементов по ключу, а также обновления, создания и удаления записей (более

полезных при просмотре таблиц, постоянно хранящихся между прогонами). Ключ в таблице (словаре) отображаемой в данный момент записи показывается в поле ввода, расположенном в середине окна.

Кнопка «index» выводит в правом верхнем углу окно списка, и выбор записи в одном из верхних окон создает окно формы внизу. Окно формы используется как для отображения записи, так и для ее редактирования – если изменить значения полей и нажать «store», то запись обновляется. Нажатие «new» очищает форму для ввода новых значений (заполните поле «Key=>» и нажмите «store», чтобы сохранить новую запись).

Значения полей вводятся согласно синтаксису Python, поэтому строки заключаются в кавычки (подробнее об этом ниже). При просмотре таблицы, записи которой содержат различные наборы имен полей, PyForm стирает и перерисовывает окно формы для новых наборов полей, когда выбираются новые записи. Чтобы не наблюдать создание окна заново, используйте одинаковый формат для всех записей данной таблицы.

Перейдем к коду. При написании PyForm я сначала создал вспомогательные функции, чтобы скрыть некоторые подробности создания графических элементов. Делая несколько упрощающих предположений (например, о протоколе упаковки), модуль примера 16.8 позволяет отделить некоторые детали кодирования GUI от остальной реализации PyForm.

Пример 16.8. PP2E\Dbase\guitools.py

```
# добавление для ширины записи, шрифт/цвет calcgui

from Tkinter import *

def frame(root, side, **extras):
    widget = Frame(root)
    widget.pack(side=side, expand=YES, fill=BOTH)
    if extras: apply(widget.config, (), extras)
    return widget

def label(root, side, text, **extras):
    widget = Label(root, text=text, relief=RIDGE)
    widget.pack(side=side, expand=YES, fill=BOTH)
    if extras: apply(widget.config, (), extras)
    return widget

def button(root, side, text, command, **extras):
    widget = Button(root, text=text, command=command)
    widget.pack(side=side, expand=YES, fill=BOTH)
    if extras: apply(widget.config, (), extras)
    return widget

def entry(root, side, linkvar, **extras):
    widget = Entry(root, relief=SUNKEN, textvariable=linkvar)
    widget.pack(side=side, expand=YES, fill=BOTH)
    if extras: apply(widget.config, (), extras)
    return widget
```

Посредством этого вспомогательного модуля файл примера 16.9 реализует остальную часть PyForm GUI. С помощью модуля `GuiMixin`, который мы написали в главе 9 «Более крупные примеры GUI», упрощается доступ к стандартным всплывающим диалогам. Он также написан как класс, который может уточняться в подклассах или прикрепляться к большим GUI. Я запускаю PyForm как самостоятельную программу. Прикрепление ее класса `FormGui` в действительности прикрепляет только главное окно, но с его помощью можно предоставить готовый графический элемент просмотра таблиц для других GUI.

Класс `FormGui` этого файла создает GUI, показанный на рис. 16.1, и отвечает на взаимодействие с пользователем во всех трех окнах интерфейса. Так как мы уже рассказывали обо всех инструментах GUI, используемых PyForm, за дополнительными деталями реализации следует обращаться к исходному коду этого модуля. Заметьте, однако, что этот файл почти ничего не знает о просматриваемой таблице, за исключением того, что она выглядит как словарь словарей. Понять, как PyForm поддерживает просмотр таких вещей, как полки экземпляров классов, можно посмотрев в другие места (или хотя бы подождав до следующего модуля).

Пример 16.9. PP2E\Dbase\TableBrowser\formgui.py

```
#!/usr/local/bin/python
###############################################################################################
# PyForm: GUI для просмотра постоянно хранящихся таблиц. Использует guimixin для стандартных
# диалогов. Предполагает наличие в просматриваемой таблице интерфейса "словарь словарей"
# и полагается на классы-оболочки для преобразования при необходимости других структур.
# Запишите начальную запись с помощью сценария dbinit, чтобы начать базу данных с нуля.
# Предостережение: не выполняет вызовы методов объектов, плохо показывает значения сложных полей.
###############################################################################################

from Tkinter  import *                                  # элементы Tk
from guitools import frame, label, button, entry        # построители графических элементов
from PP2E.Gui.Tools.guimixin import GuiMixin            # общие методы

class FormGui(GuiMixin, Frame):
    def __init__(self, mapping):                       # расширенный фрейм
        Frame.__init__(self)                            # на верхнем уровне по умолчанию
        self.pack(expand=YES, fill=BOTH)                # все части расширяемы
        self.master.title('PyForm 2.0 - Table browser')
        self.master.iconname("PyForm")
        self.makeMainBox()
        self.table     = mapping                        # словарь, dbm, полка, таблица...
        self.index     = mapping.keys()                 # список ключей таблицы
        self.cursor    = -1                             # текущее положение индекса
        self.currslots = []                             # (key,text) текущей формы
        self.currform  = None                           # окно текущей формы
        self.listbox   = None                           # окно списка указателя

    def makeMainBox(self):
        frm = frame(self, TOP)
        frm.config(bd=2)
        button(frm, LEFT, 'next',  self.onNext)         # следующий по списку
        button(frm, LEFT, 'prev',  self.onPrev)         # предыдущий по списку
        button(frm, LEFT, 'find',  self.onFind)         # поиск по ключу
        frm = frame(self, TOP)
        self.keytext = StringVar()                      # ключ текущей записи
        label(frm, LEFT, 'KEY=>')                       # изменить перед 'find'
        entry(frm, LEFT,  self.keytext)
        frm = frame(self, TOP)
        frm.config(bd=2)
        button(frm,  LEFT,  'store',  self.onStore)     # обновленные данные записи
        button(frm,  LEFT,  'new',    self.onNew)       # очистить поля
        button(frm,  LEFT,  'index',  self.onMakeList)  # показать список ключей
        button(frm,  LEFT,  'delete', self.onDelete)    # показать список ключей
        button(self, BOTTOM,'quit',   self.quit)        # из guimixin

    def onPrev(self):
        if self.cursor <= 0:
            self.infobox('Backup', "Front of table")
```

```
        else:
            self.cursor = self.cursor - 1
            self.display()

    def onNext(self):
        if self.cursor >= len(self.index)-1:
            self.infobox('Advance', "End of table")
        else:
            self.cursor = self.cursor + 1
            self.display()

    def sameKeys(self, record):                     # можно повторно использовать ту же форму?
        keys1 = record.keys()
        keys2 = map(lambda x:x[0], self.currslots)
        keys1.sort(); keys2.sort()                  # порядок списка ключей другой
        return keys1 == keys2                       # если порядок вставки другой

    def display(self):
        key = self.index[self.cursor]               # показать запись по курсору указателя
        self.keytext.set(key)                       # изменить ключ в главном окне
        record = self.table[key]                    # в словаре, dbm, полке, классе
        if self.sameKeys(record):
            self.currform.title('PyForm - Key=' + `key`)
            for (field, text) in self.currslots:
                text.set(`record[field]`)           # те же поля? повторно использовать форму
        else:                                       # `x` действует как repr(x)
            if self.currform:
                self.currform.destroy()             # разные поля?
            new = Toplevel()                        # заменить текущее окно
            new.title('PyForm - Key=' + `key`)      # новое растягиваемое окно
            new.iconname("pform")
            left  = frame(new, LEFT)
            right = frame(new, RIGHT)
            self.currslots = []                     # список (поле, значение)
            for field in record.keys():
                label(left, TOP, `field`)           # ключ,значение в строки
                text = StringVar()                  # можно было бы отсортировать здесь ключи
                text.set( `record[field]` )
                entry(right, TOP, text, width=40)
                self.currslots.append((field, text))
            self.currform = new
            new.protocol('WM_DELETE_WINDOW', lambda:0)   # игнорировать destroy
        self.selectlist()                                # обновить окно списка

    def onStore(self):
        if not self.currform: return
        key = self.keytext.get()
        if key in self.index:                       # изменить имеющуюся запись
            record = self.table[key]                # не self.table[key][field]=
        else:
            record = {}                             # создать новую запись
            self.index.append(key)                  # добавить в указатель и окно списка
            if self.listbox:
                self.listbox.insert(END, key)       # или в len(self.index)-1
        for (field, text) in self.currslots:
            try:                                    # заполнить запись словаря
                record[field] = eval(text.get())    # преобразовать обратно в строку
            except:
                self.errorbox('Bad data: "%s" = "%s"' % (field, text.get()))
```

```
            record[field] = None
        self.table[key] = record                    # добавить в словарь, dbm, полку...
        self.onFind(key)                            # считывание: установить курсор, окно списка

    def onNew(self):
        if not self.currform: return                # очистить входную форму и ключ
        self.keytext.set('?%d' % len(self.index)) # ключ по умолчанию, если не введен
        for (field, text) in self.currslots:        # очистить ключ/поля для ввода
            text.set('')
        self.currform.title('Key: ?')

    def onFind(self, key=None):
        target = key or self.keytext.get()            # передан или введен
        try:
            self.cursor = self.index.index(target)   # найти метку в  списке ключей
            self.display()
        except:
            self.infobox('Not found', "Key doesn't exist", 'info')

    def onDelete(self):
        if not self.currform or not self.index: return
        currkey = self.index[self.cursor]
        del self.table[currkey]                     # таблица, указатель на окно списка
        del self.index[self.cursor:self.cursor+1] # как "list[i:i+1] = []"
        if self.listbox:
            self.listbox.delete(self.cursor)        # удалить из окна списка
        if self.cursor < len(self.index):
            self.display()                          # показать следующую запись, если она есть
        elif self.cursor > 0:
            self.cursor = self.cursor-1             # показать предыдущую, если удаление с конца
            self.display()
        else:                                     # оставить окно, если удаляется последняя запись
            self.onNew()

    def onList(self,evnt):
        if not self.index: return                   # при двойном щелчке по окну списка
        index = self.listbox.curselection()         # выбрать текст выделенного ключа
        label = self.listbox.get(index)             # или listbox.get(ACTIVE)
        self.onFind(label)                          # и вызвать метод

    def onMakeList(self):
        if self.listbox: return                     # уже есть?
        new = Toplevel()                            # новое растягиваемое окно
        new.title("PyForm - Key Index")             # выбор ключей из окна списка
        new.iconname("pindex")
        frm    = frame(new, TOP)
        scroll = Scrollbar(frm)
        list   = Listbox(frm, bg='white')
        scroll.config(command=list.yview, relief=SUNKEN)
        list.config(yscrollcommand=scroll.set, relief=SUNKEN)
        scroll.pack(side=RIGHT, fill=BOTH)
        list.pack(side=LEFT, expand=YES, fill=BOTH)    # упакован последним, обрезается первым
        for key in self.index:                         # добавить в окно списка
            list.insert(END, key)                      # или: сначала сортировать список
        list.config(selectmode=SINGLE, setgrid=1)      # режимы выбора, растяжки
        list.bind('<Double-1>', self.onList)           # по двойному щелчку
        self.listbox = list
        if self.index and self.cursor >= 0:            # выделить положение
            self.selectlist()
```

```
        new.protocol('WM_DELETE_WINDOW', lambda:0)     # игнорировать destroy

    def selectlist(self):                              # список следит за курсором
        if self.listbox:
            self.listbox.select_clear(0, self.listbox.size())
            self.listbox.select_set(self.cursor)

if __name__ == '__main__':
    from PP2E.Dbase.testdata import cast        # код самотестирования
    for k in cast.keys(): print k, cast[k]      # просмотр словаря словарей из памяти
    FormGui(cast).mainloop()
    for k in cast.keys(): print k, cast[k]      # при выходе показать модифицированную таблицу
```

Код самотестирования файла запускает PyForm GUI для просмотра располагаемого в памяти словаря словарей `cast` из модуля `testdata`, который был приведен выше. Для запуска PyForm создается и выполняется объект класса `FormGui`, определяемый в этом файле, которому передается таблица для просмотра. Ниже показаны сообщения, появляющиеся в `stdout` после выполнения этого файла и редактирования нескольких записей, показываемых в GUI; словарь показывается при запуске GUI и выходе из него:

```
C:\...\PP2E\Dbase\TableBrowser>python formgui.py
alan {'job': 'comedian', 'name': ('Alan', 'B')}
sally {'job': 'writer', 'name': ('Sally', 'R')}
rob {'spouse': 'Laura', 'job': 'writer', 'name': ('Rob', 'P')}
mel {'job': 'producer', 'name': ('Mel', 'C')}
milly {'spouse': 'Jerry', 'name': ('Milly', '?'), 'kids': 2}
buddy {'spouse': 'Pickles', 'job': 'writer', 'name': ('Buddy', 'S')}
laura {'spouse': 'Rob', 'name': ('Laura', 'P'), 'kids': 1}

alan {'job': 'comedian', 'name': ('Alan', 'B')}
jerry {'spouse': 'Milly', 'name': 'Jerry', 'kids': 0}
sally {'job': 'writer', 'name': ('Sally', 'R')}
rob {'spouse': 'Laura', 'job': 'writer', 'name': ('Rob', 'P')}
mel {'job': 'producer', 'name': ('Mel', 'C')}
milly {'spouse': 'Jerry', 'name': ('Milly', '?'), 'kids': 2}
buddy {'spouse': 'Pickles', 'job': 'writer', 'name': ('Buddy', 'S')}
laura {'name': ('Laura', 'P'), 'kids': 3, 'spouse': 'bob'}
```

Последняя строка (полужирная) показывает изменения, сделанные в GUI. Так как эта таблица располагается в памяти, изменения, проведенные в GUI, не сохраняются (сами по себе словари не являются постоянными). Чтобы посмотреть, как использовать PyForm GUI с постоянными хранилищами типа файлов DBM и полок, перейдем к следующей теме.

Оболочки таблиц PyForm

В следующем файле определены общие классы, служащие «оболочками» для различных типов таблиц (взаимодействующие с ними) в PyForm. Благодаря им становится возможным использование PyForm с разнообразными таблицами.

Предыдущий модуль был написан для выполнения разных действий в GUI в предположении, что таблицы открывают интерфейс в виде словаря словарей. Напротив, следующему модулю ничего не известно о GUI, но он выполняет трансляцию, необходимую для просмотра в PyForm объектов, отличных от словарей. В действительности этот модуль даже не импортирует Tkinter – он занят строго преобразованием протокола объектов и ничем другим. Благодаря разделению реализации PyForm на такие отдельные функциональные модули легче сосредоточиться на задаче, выполняемой каждым модулем.

Связь между двумя модулями устроена так: для таблиц специального вида в FormGui PyForm передается экземпляр класса Table, код которого здесь находится. Класс Table перехватывает операции выборки по индексу и присвоения и с помощью встроенного класса оболочки записи при необходимости преобразует записи в формат словаря и обратно.

Например, поскольку файлы DBM могут хранить только строки, Table осуществляет преобразование между действительными словарями и их строковым представлением при сохранении или загрузке таблицы. Для экземпляров классов Table извлекает словарь атрибутов объекта __dict__ при выборке и копирует поля словаря в атрибуты вновь созданного экземпляра класса при сохранении.[1] В итоге GUI представляет себе все таблицы как словари, даже если в действительности они являются чем-то совсем иным.

При изучении листинга этого модуля, показанного в примере 16.10, обратите внимание на отсутствие в нем данных о формате записи в какой-либо конкретной базе данных. На самом деле их не было и в модуле formgui, связанном с GUI. Поскольку ни один из модулей не интересуется структурой полей в записях базы данных, оба они могут использоваться для просмотра произвольных записей.

Пример 16.10. PP2E\Dbase\formtable.py

```
##############################################################################################
# Классы-оболочки таблиц и тесты PyForm. Поскольку PyForm предполагает интерфейс словаря
# словарей, этот модуль преобразует записи строк и экземпляров классов в словари и обратно.
# PyForm содержит преобразование таблиц--Table не является подклассом PyForm. Обратите
# внимание, что некоторые классы-оболочки могут использоваться вне PyForm - Dbm опечатка
# в оригинале OfString может быть оболочкой dbm с данными произвольных типов. Выполните
# сценарии dbinit, чтобы начать новую базу данных, и выполните dbview для просмотра другой базы
# данных вместо той, которая здесь проверяется. Больше не требуется, чтобы у классов
# были значения по умолчанию аргументов конструктора, автоматически выбирается класс записи
# из первой загруженной, если не передан в класс-оболочку записи. Предупреждение: все же
# предполагается, что все экземпляры в таблице принадлежат одному классу.
##############################################################################################

##############################################################################################
# записи в таблицах
##############################################################################################

class DictionaryRecord:
    def todict(self, value):
        return value                    # в словарь: не нужно преобразовывать
    def fromdict(self, value):
        return value                    # из словаря: не нужно преобразовывать

class StringRecord:
    def todict(self, value):
        return eval(value)              # преобразовать строку в словарь (или другое)
    def fromdict(self, value):
        return str(value)               # преобразовать словарь (или другое) в строку

class InstanceRecord:
    def __init__(self, Class=None):         # нужен объект класса для создания экземпляров
```

[1] Еще одна тонкая вещь: подобно новому модулю pickle PyForm пытается генерировать новый экземпляр класса в операциях сохранения, просто заставляя атрибут __class__ общего объекта экземпляра указывать на исходный класс; если это не удается, PyForm возвращается к вызову класса без аргументов (тогда у класса должны быть значения по умолчанию для аргументов конструктора, отличные от «self»). Присвоение значения __class__ может быть не выполнено в режиме ограниченного выполнения. Дополнительные сведения можно найти в классе InstanceRecord в листинге исходного кода.

```
        self.Class = Class
    def todict(self, value):                # преобразовать экземпляр в словарь атрибутов
        if not self.Class:                  # получить класс из объекта, если он не известен
            self.Class = value.__class__
        return value.__dict__
    def fromdict(self, value):              # преобразовать словарь атрибутов в экземпляр
        try:
            class Dummy: pass               # проверить новую сериализацию
            instance = Dummy()              # отказ в ограниченном режиме
            instance.__class__ = self.Class
        except:                             # иначе вызов класса без аргументов
            instance = self.Class()         # нужны значения по умолчанию
        for attr in value.keys():
            setattr(instance, attr, value[attr])  # установить атрибуты экземпляра
        return instance                           # может выполнить Class.__setattr__

################################################################################
# таблица с записями
################################################################################

class Table:
    def __init__(self, mapping, converter):   # объект таблица, преобразователь записей
        self.table  = mapping                  # оболочка произвольного преобразования таблицы
        self.record = converter                # оболочка произвольного типа записи

    def storeItems(self, items):               # инициализация из словаря
        for key in items.keys():               # выполнить __setitem__ для трансляции, записи
            self[key] = items[key]

    def printItems(self):                      # вывод соответствия в оболочке
        for key in self.keys():                # self.keys получает ключи таблицы
            print key, self[key]               # __getitem__ выбирает, транслирует

    def __getitem__(self, key):                # при загрузке по индексу tbl[key]
        rawval = self.table[key]               # загрузить из таблицы соответствия
        return self.record.todict(rawval)      # транслировать в словарь

    def __setitem__(self, key, value):         # при присвоении по индексу tbl[key]=val
        rawval = self.record.fromdict(value)   # транслировать из словаря
        self.table[key] = rawval               # сохранить в таблице соответствия

    def __delitem__(self, key):                # удалить из таблицы соответствия
        del self.table[key]

    def keys(self):                           # получить указатель ключей преобразования таблицы
        return self.table.keys()

    def close(self):
        if hasattr(self.table, 'close'):      # вызвать закрытие таблицы, если оно определено
            self.table.close()                # может требоваться для полок, dbm

################################################################################
# комбинации таблица/запись
################################################################################

import shelve, anydbm

def ShelveOfInstance(filename, Class=None):
    return Table(shelve.open(filename), InstanceRecord(Class))
def ShelveOfDictionary(filename):
    return Table(shelve.open(filename), DictionaryRecord())
def ShelveOfString(filename):
    return Table(shelve.open(filename), StringRecord())
```

```
def DbmOfString(filename):
    return Table(anydbm.open(filename, 'c'), StringRecord())

def DictOfInstance(dict, Class=None):
    return Table(dict, InstanceRecord(Class))
def DictOfDictionary(dict):
    return Table(dict, DictionaryRecord())
def DictOfString(filename):
    return Table(dict, StringRecord())

ObjectOfInstance   = DictOfInstance           # другие отображаемые объекты
ObjectOfDictionary = DictOfDictionary         # классы, похожие на словари
ObjectOfString     = DictOfString

#############################################################################
# проверка обычного применения
#############################################################################

if __name__ == '__main__':
    from sys import argv
    from formgui import FormGui                         # получить gui для словаря
    from PP2E.Dbase.testdata import Actor, cast         # получить класс, словарь словарей
    TestType   = 'shelve'                               # полка, dbm, словарь
    TestInit   = 0                                      # инициализировать файл при запуске?
    TestFile   = '../data/shelve1'                      # внешний файл
    if len(argv) > 1: TestType = argv[1]
    if len(argv) > 2: TestInit = int(argv[2])
    if len(argv) > 3: TestFile = argv[3]

    if TestType == 'shelve':                            # python formtbl.py shelve?
        print 'shelve-of-instance test'
        table = ShelveOfInstance(TestFile, Actor)       # заключить полку в объект Table
        if TestInit:
            table.storeItems(cast)                      # python formtbl.py shelve 1
        FormGui(table).mainloop()
        table.close()
        ShelveOfInstance(TestFile).printItems()         # класс, выбираемый при загрузке
    elif TestType == 'dbm':                             # python formtbl.py dbm
        print 'dbm-of-dictstring test'
        table = DbmOfString(TestFile)                   # заключить dbm в объект Table
        if TestInit:
            table.storeItems(cast)                      # python formtbl.py dbm 1
        FormGui(table).mainloop()
        table.close()
        DbmOfString(TestFile).printItems()              # вывести содержимое новой таблицы
```

Кроме классов `Table` и оболочки записи в модуле определены функции генерации (например, `ShelveOfInstance`), создающие `Table` для всех мыслимых комбинаций таблицы и записи. Не все комбинации разрешены; например файлы DBM могут содержать только словари, закодированные в виде строк, потому что экземпляры классов не просто отобразить в формат строковых значений, ожидаемый DBM. Однако эти классы достаточно гибки, чтобы позволить ввести дополнительные конфигурации `Table`.

Единственное, что связано с GUI в этом файле, – это находящийся в конце код самотестирования. При запуске в качестве сценария этот модуль запускает PyForm GUI для просмотра или редактирования полки или экземпляров постоянно хранящегося класса `Actor` или файла DBM со словарями путем передачи правильного типа объекта `Table`. GUI выглядит так же, как показанный на рис. 16.1; при запуске без аргументов код самотестирования позволяет просматривать полку экземпляров классов:

```
C:\...\PP2E\Dbase\TableBrowser>python formtable.py
shelve-of-instance test
...вывод содержимого при выходе...
```

Поскольку PyForm выводит в этом случае полку, все сделанные изменения сохраняются после выхода из GUI. Чтобы снова инициализировать полку из словаря cast в `testdata`, передайте «1» в качестве второго аргумента («0» означает отсутствие повторной инициализации). Чтобы заменить установленное в сценарии по умолчанию имя файла полки, передайте новое имя в качестве третьего аргумента:

```
C:\...\PP2E\Dbase\TableBrowser>python formtable.py shelve 1
C:\...\PP2E\Dbase\TableBrowser>python formtable.py shelve 0 ../data/shelve1
```

Чтобы проверить PyForm на файле DBM со словарями, преобразованными в строки, передайте `dbm` в качестве первого аргумента командной строки; следующие два аргумента действуют так же:

```
C:\...\PP2E\Dbase\TableBrowser>python formtable.py dbm 1 ..\data\dbm1
dbm-of-dictstring test
...вывод содержимого при выходе...
```

Наконец, поскольку эти самопроверки в конечном итоге обрабатывают конкретные файлы полок и DBM, можно вручную открыть и изучить их содержимое с помощью обычных библиотечных вызовов. Вот как они выглядят при открытии в интерактивном сеансе:

```
C:\...\PP2E\Dbase\data>ls
dbm1        myfile         shelve1

C:\...\PP2E\Dbase\data>python
>>> import shelve
>>> db = shelve.open('shelve1')
>>> db.keys()
['alan', 'buddy', 'sally', 'rob', 'milly', 'laura', 'mel']
>>> db['laura']
<PP2E.Dbase.testdata.Actor instance at 799850>

>>> import anydbm
>>> db = anydbm.open('dbm1')
>>> db.keys()
['alan', 'mel', 'buddy', 'sally', 'rob', 'milly', 'laura']
>>> db['laura']
"{'name': ('Laura', 'P'), 'kids': 2, 'spouse': 'Rob'}"
```

Файл полки содержит реальные объекты экземпляров класса `Actor`, а файл DBM содержит словари, преобразованные в строки. Оба формата сохраняются в этих файлах между запусками GUI и преобразуются обратно в словари, чтобы быть снова показанными в дальнейшем.[1]

---

[1] Обратите внимание, что файлы DBM, содержащие словари, используют `str` и `eval` для преобразования в строки и обратно, но могли бы записывать в файлы DBM сериализованные с помощью `pickle` представления словарей записей. Но так как этим в точности занимается `shelve`, для иллюстрации здесь была выбрана схема `str/eval`. Предлагаемое упражнение: добавить новый класс записей `PickleRecord`, основанный на функциях `loads` и `dumps` модуля `pickle`, описанных ранее в этой главе, и сравнить его по производительности с `StringRecord`. Смотрите также структуру базы данных файла `pickle` в главе 14; схема его каталога с одним плоским файлом для каждой записи могла бы также использоваться здесь для реализации «таблицы» при создании надлежащих подклассов `Table`.

Вспомогательные сценарии PyForm для создания и просмотра

Код самопроверки модуля formtable показывает его работоспособность, но ограничивается файлами и классами готовых контрольных примеров. А нельзя ли использовать PyForm с другими типами баз данных, хранящими более полезные данные?

К счастью, модули formgui и formtable были написаны с учетом общности – они не зависят от формата записей какой-либо конкретной базы данных. Благодаря этому можно легко направить PyForm на собственные базы данных: импортируйте и запустите объект FormGui с таблицей (возможно, заключенной в оболочку), которую нужно просмотреть.

Необходимые для начального запуска вызовы не очень сложны, и можно вводить их в интерактивной подсказке каждый раз, когда нужно просмотреть базу данных; но обычно проще хранить их для повторного использования в сценариях. Скажем, сценарий примера 16.11 можно запускать для открытия PyForm с *любой* полкой, содержащей записи, хранящиеся в формате экземпляра класса или словаря.

Пример 16.11. PP2E\Dbase\dbview.py

```
##############################################################################################
# непосредственный просмотр любых полок; более общий случай, чем командная строка
# "formtable.py shelve 1 filename", действительная только для Actor;
# передайте в параметрах имя файла (и режим), чтобы просмотреть любую полку:
# formtable автоматически выбирает класс из первого загруженного экземпляра;
# запустите dbinit1 для (ре)инициализации полки базы данных по шаблону.
##############################################################################################

from sys import argv
from formtable import *
from formgui import FormGui

mode = 'class'
file = '../data/mydbase-' + mode
if len(argv) > 1: file = argv[1]      # dbview.py файл? режим??
if len(argv) > 2: mode = argv[2]

if mode == 'dict':
    table = ShelveOfDictionary(file)  # просмотр словарей
else:
    table = ShelveOfInstance(file)    # просмотр объектов классов

FormGui(table).mainloop()
table.close()                         # некоторые dbm требуется закрывать
```

Единственная неприятность здесь в том, что PyForm не очень хорошо обрабатывает совершенно *пустые* таблицы; нельзя добавлять в GUI новые записи, если никаких записей еще нет. То есть в PyForm нет средства разработки структуры записи: кнопка «new» просто очищает имеющуюся форму ввода.

По этой причине, чтобы создать новую базу данных с пустого места, требуется добавить начальную запись, которая даст PyForm структуру поля. Для этого требуется только несколько строк кода, которые можно ввести интерактивно, но почему бы не поместить их в обобщенные сценарии для повторного использования? Файл примера 16.12 показывает один из способов инициализации базы данных PyForm с первой пустой записью.

Пример 16.12. PP2E\Dbase\dbinit1.py

```
##############################################################################################
# Сохранить первую запись на новой полке, чтобы создать начальный список полей;
# PyForm GUI требует существования записей, чтобы можно было добавлять новые; удалите запись
```

```
# шаблона с '?' после добавления реальных записей; измените режим, файл, шаблон
# для использования с данными других типов; при заполнении полок из других файлов данных этого
# не требуется; см. dbinit2 для основанной на объектах версии и dbview для просмотра полок.
#########################################################################################

import os
from sys import argv
mode = 'class'
file = '../data/mydbase-' + mode
if len(argv) > 1: file = argv[1]                    # dbinit1.py файл? режим??
if len(argv) > 2: mode = argv[2]
try:
    os.remove(file)                                 # удалить имеющийся
except: pass

if mode == 'dict':
    template = {'name': None, 'age': None, 'job': None}   # начать полку словаря
else:
    from PP2E.Dbase.person import Person            # один аргумент со значением по умолчанию
    template = Person(None, None)                   # начать полку объектов

import shelve
dbase = shelve.open(file)                           # создать сейчас
dbase['?empty?'] = template
dbase.close()
```

Теперь просто измените какие-то установки в этом сценарии или задайте аргументы командной строки, чтобы создать новую базу данных, основанную на полках для использования в PyForm. Можно заменить любой список полей или имя класса в этом сценарии, чтобы сопровождать с помощью PyForm простую объектную базу данных для хранения реальной информации (две такие базы данных мы увидим через мгновение). Пустая запись появляется с ключом «?empty?» при первом просмотре базы данных с помощью `dbview`; замените ее первой реальной записью с помощью клавиши PyForm «store» – и все в порядке. Если не изменять полку базы данных, кроме как в этом GUI, у всех ее записей будет одинаковый формат полей, который определен в сценарии инициализации.

Но обратите внимание, что сценарий `dbinit1` сразу обращается к файлу полки для сохранения первой записи; сегодня это хорошо, но может рухнуть, если PyForm изменить, чтобы он делал что-то более специальное с представлением хранящихся данных. Видимо, лучший способ заполнить таблицы вне GUI, – это использовать классы-оболочки `Table`, применяемые в нем. Например, следующий альтернативный сценарий инициализирует базу данных PyForm с помощью генерируемых объектов `Table`, а не прямыми действиями с полками (см. пример 16.13).

Пример 16.13. PP2E\Dbase\dbinit2.py

```
#########################################################################
# Этот код тоже работает – на базе объектов Table, а не ручных операций с полками;
# сохранить первую запись в полке, как требует PyForm GUI.
#########################################################################

from formtable import *
import sys, os

mode = 'dict'
file = '../data/mydbase-' + mode
if len(sys.argv) > 1: file = sys.argv[1]
if len(sys.argv) > 2: mode = sys.argv[2]
```

```
try:
    os.remove(file)
except: pass

if mode == 'dict':
    table    = ShelveOfDictionary(file)
    template = {'name': None, 'shoesize': None, 'language': 'Python'}
else:
    from PP2E.Dbase.person import Person
    table    = ShelveOfInstance(file, Person)
    template = Person(None, None).__dict__

table.storeItems({'?empty?': template})
table.close()
```

Применим эти сценарии для инициализации и редактирования двух пользовательских баз данных. Рис. 16.2 иллюстрирует просмотр одной из них после инициализации базы данных с помощью сценария и добавления ряда реальных записей через GUI.

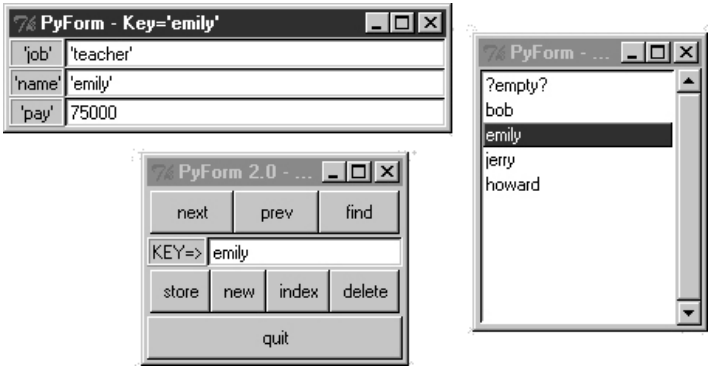


Рис. 16.2. Полка с объектами Person (dbinit1, dbview)

Здесь окно списка показывает запись, которую я добавил на полку в GUI. Для инициализации базы данных с помощью начальной записи и открытия ее в PyForm для добавления записей (то есть экземпляров класса `Person`) я выполнил следующие команды:

```
C:\...\PP2E\Dbase\TableBrowser>python dbinit1.py

C:\...\PP2E\Dbase\TableBrowser>python dbview.py
```

Можно изменить имя класса и словарь полей в сценариях `dbinit`, чтобы инициализировать записи для любого типа баз данных, которые вы собираетесь сопровождать с помощью PyForm. Используйте словари, если не хотите представлять постоянные объекты с помощью классов (но классы позволяют добавлять другие виды поведения как методы, не видимые в PyForm). Следите за тем, чтобы использовать отдельное имя файла для каждой базы данных; начальную запись «?empty?» можно удалить после добавления реальной записи (позднее просто выделите запись в окне списка и нажмите «new», чтобы очистить форму для ввода новых значений записи).

Данные, отображаемые в GUI, представляют подлинную полку постоянных объектов экземпляров класса `Person` – изменения и добавления, осуществляемые в GUI, будут сохранены до следующего раза, когда вы будете просматривать эту полку с помощью PyForm. Если вы любите работать с клавиатурой, то можете открыть полку напрямую, чтобы проверить работу PyForm:

```
C:\...\PP2E\Dbase\data>ls
mydbase-class  myfile         shelve1

C:\...\PP2E\Dbase\data>python
>>> import shelve
>>> db = shelve.open('mydbase-class')
>>> db.keys()
['emily', 'jerry', '?empty?', 'bob', 'howard']
>>> db['bob']
<PP2E.Dbase.person.Person instance at 798d70>
>>> db['emily'].job
'teacher'
>>> db['bob'].tax
30000.0
```

Обратите внимание, что «bob» является экземпляром класса `Person`, с которым мы встречались выше (см. раздел о полках). В предположении, что это та же версия модуля `person`, в которой был введен метод `__getattr__`, запрос атрибута `tax` объекта с полки вычисляет значение на лету, так как в действительности он вызывает *метод* класса. Заметьте, что это действует несмотря на то, что `Person` здесь не импортировался – Python загружает класс внутренне при воссоздании хранящихся на полке экземпляров.

Так же легко основать совместимую с PyForm базу данных на внутренней структуре словаря, а не классах. На рис. 16.3 показан просмотр такой базы данных после инициализации с помощью сценария и заполнения через GUI.

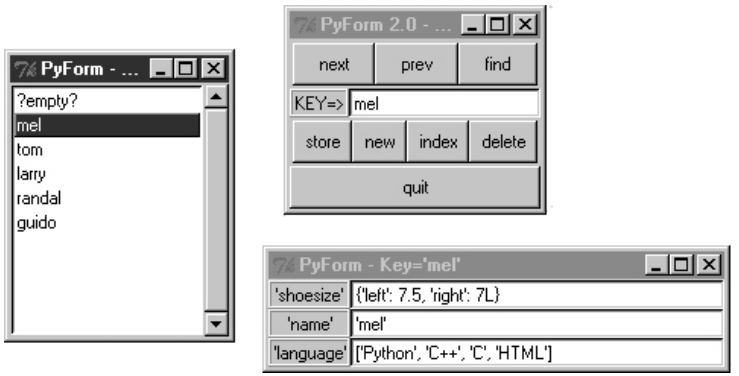


Рис. 16.3. Полка словарей (dbinit2, dbview)

Помимо другого внутреннего формата у этой базы другая структура записи (имена полей ее записей не такие, как в последнем примере), и она хранится в собственном файле полки. Вот команды, с помощью которых я инициализировал и редактировал эту базу данных:

```
C:\...\PP2E\Dbase\TableBrowser>python dbinit2.py ../data/mydbase-dict dict

C:\...\PP2E\Dbase\TableBrowser>python dbview.py ../data/mydbase-dict dict
```

Добавив на полку несколько записей (то есть *словарей*), можно снова просматривать их в PyForm или открыть полку вручную, чтобы проверить работу PyForm:

```
C:\...\PP2E\Dbase\data>ls
mydbase-class  mydbase-dict   myfile         shelve1

C:\...\PP2E\Dbase\data>python
>>> db = shelve.open('mydbase-dict')
```

```
>>> db.keys()
['tom', 'guido', '?empty?', 'larry', 'randal', 'mel']
>>> db['guido']
{'shoesize': 42, 'name': 'benevolent dictator', 'language': 'Python'}
>>> db['mel']['shoesize']
{'left': 7.5, 'right': 7L}
```

На этот раз записи на полке действительно являются словарями, а не экземплярами класса или преобразованными строками. Для PyForm это, однако, безразлично: поскольку все таблицы заключаются в оболочки, чтобы соответствовать интерфейсу PyForm, оба формата одинаково выглядят при просмотре в GUI.

Заметьте, что поля «shoe size» и «language» на этом снимке экрана в действительности являются словарем и списком. Чтобы задать значения, можно ввести в поля формы этого GUI синтаксис любого выражения Python (поэтому строки заключены здесь в кавычки). PyForm использует выражение Python обратных кавычек, чтобы преобразовать объекты значений отображения (`x` похоже на `repr(x)`, которая похожа на `str(x)`, но с добавлением кавычек вокруг строк). Для преобразования из строк обратно в объекты значений PyForm использует функцию Python `eval`, которая анализирует и вычисляет код, введенный в поля. Поле ввода/отображения ключа в главном окне не добавляет и не принимает кавычек вокруг строки ключа, потому что ключи все же должны быть строками в таких вещах, как полки (хотя поля могут иметь произвольный тип).

Как мы неоднократно отмечали, функция `eval` (и родственная ей `exec`) является мощной, но опасной – никогда нельзя быть уверенным, что введенный пользователем код не удалит файлы, не подвесит систему, не напишет e-mail вашему боссу и т. д. Если нет уверенности, что значения полей не будут содержать вредоносный код (по злому умыслу или нет), используйте для вычисления строк средства режима ограниченного выполнения `rexec`, с которыми мы познакомились в главе 15. Другим вариантом является простое ограничение типов разрешенных операций и вычисление их более простыми средствами (например, `int`, `str`, `string.atoi`).

Хотя PyForm предполагает найти в просматриваемых таблицах интерфейс (протокол) словаря словарей, этой форме соответствует удивительно большое число объектов, поскольку словари вездесущи в объектах Python. В действительности PyForm можно использовать для просмотра того, что не имеет никакого отношения к понятиям таблиц или записей баз данных, но может удовлетворять протоколу.

Например, таблица Python `sys.modules`, которую мы встретили в главе 2, является встроенным словарем объектов загруженных модулей. Если с помощью подходящего класса оболочки заставить модули быть похожими на словари, нет причин, препятствующих просмотру `sys.modules` из памяти с помощью PyForm, как показано в примере 16.14.

Пример 16.14. PP2E\Dbase\TableBrowser\viewsysmod.py

```
# просмотр таблицы sys.modules в FormGui

class modrec:
    def todict(self, value):
        return value.__dict__      # не dir(value): нужен словарь
    def fromdict(self, value):
        assert 0, 'Module updates not supported'
```

```
import sys
from formgui import FormGui
from formtable import Table
FormGui(Table(sys.modules, modrec())).mainloop()
```

Этот сценарий определяет класс для извлечения словаря атрибутов модуля `__dict__` (`InstanceRecord` из `formtable` не годится, потому что она ищет также `__class__`). В основном он просто передает `sys.modules` в PyForm (`FormGui`), заключенный в объект `Table`; результат показан на рис. 16.4.

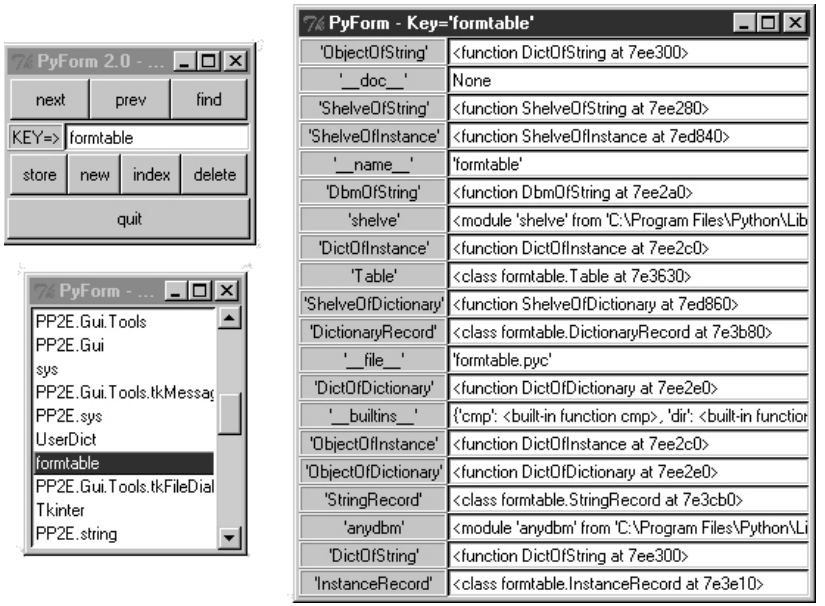


Рис. 16.4. FormGui просматривает sys.modules (viewsysmod)

С помощью аналогичных оболочек записей и таблиц можно просматривать в PyForm объекты любого вида. Как обычно в Python, значение имеет только то, что они обеспечивают совместимый интерфейс.

Ограничения PyForm

Хотя сценарий просмотра `sys.modules` работает, он выявляет ряд ограничений действующей архитектуры PyForm:

Только два уровня

Устройство PyForm позволяет обрабатывать только двумерную структуру отображения таблица/запись. Нельзя опуститься ниже в поля, показываемые на форме, большие структуры данных в полях выводятся как длинные строки, а сложные объекты типа вложенных модулей, классов и функций, содержащих собственные атрибуты, показываются в представлении, установленном для вывода по умолчанию. Можно было бы добавить средства просмотра объектов для интерактивного изучения вложенных объектов, но их может оказаться сложно написать.

Нет поддержки больших форм

PyForm не приспособлен для обработки большого количества полей записей – если выбрать в окне списка запись для модуля `os` (см. рис. 16.4), то получится гигант-

ская форма, которая может не уместиться на экране (у модуля `os` масса атрибутов; на моем экране их умещается только около 40). Можно исправить положение, добавив полосу прокрутки, хотя маловероятно, что в базах данных, для просмотра которых создан PyForm, записи будут содержать десятки полей.

Вывод только атрибутов данных

PyForm выводит значения атрибутов записей, но не поддерживает вызов функций методов просматриваемых объектов и не может выводить динамически вычисляемые атрибуты (например, атрибут `tax` в объектах `Person`).

Только один класс в таблице

PyForm в настоящее время предполагает, что все экземпляры в таблице относятся к одному классу, хотя этого не требуется для полок в целом.

Иными словами, если хотите поэкспериментировать, здесь есть над чем поработать. Можно также попробовать другие стили программирования. Например, PyForm в настоящее время перегружает выборку и присвоение по индексу *таблицы*, а GUI использует словари для внутреннего представления записей. Почти так же просто было бы вместо этого перегружать выборку и присвоение по индексу поля *записи* и добавить в `Table` метод создания новой пустой записи. При такой схеме записи в PyForm будут теми объектами, которые хранятся в таблице (не словарями), а выборка или присвоение для каждого поля в PyForm будут пересылаться классам оболочек записей. Недостаток такого подхода в том, что PyForm не сможет просматривать объекты, для которых нет оболочки в `Table`. Просто словари не будут работать, потому что в них нет методов для создания новых пустых записей. Более того, для файлов DBM, в которых целые записи отображаются в строки, может потребоваться дополнительная логика для обработки запросов одиночных полей.

С другой стороны, расширения в этой области не ограничены, поэтому предлагаем их в качестве упражнения. Не предполагалось делать из PyForm общее средство просмотра объектов Python. Но в качестве простого интерфейса GUI к таблицам постоянных объектов он решает поставленные перед ним задачи. Полки и классы Python облегчают программирование таких систем и наделяют их мощью. Сложные данные можно сохранять и выбирать за один шаг, а также добавлять к ним методы, обеспечивающие динамическое поведение записей. Дополнительным преимуществом является то, что при написании таких программ на Python с Tkinter они автоматически становятся переносимыми между большинством платформ GUI. При соединении методов сохранения Python с GUI множество функций оказывается получено «даром».

17

Структуры данных

«Розы – красные, фиалки – голубые; списки изменяемы, так же как класс Foo»

Структуры данных являются сердцевиной большинства программ, даже когда это незаметно. Это не всегда очевидно, потому что Python предоставляет ряд встроенных типов, облегчающих работу со структурированными данными: списки, строки, кортежи, словари и т. п. В простых системах этих типов обычно достаточно. Технически словари делают многие классические алгоритмы поиска ненужными в Python, а списки избавляют от большой части работы, необходимой для поддержки совокупностей в языках более низкого уровня. Теми и другими настолько просто пользоваться, однако, что обычно не приходится задумываться над ними.

Но в более развитых приложениях может потребоваться ввести собственные более сложные типы, чтобы справиться с дополнительными требованиями. В этой главе мы рассмотрим несколько реализаций в Python более продвинутых структур данных: множеств, стеков, графов и т. д. Как будет показано, структуры данных принимают в Python вид новых типов объектов, интегрированных с моделью типов языка. То есть объекты, которые мы кодируем на Python, становятся законченными типами данных – для использующих их сценариев они выглядят точно так же, как списки, числа и словари.

Хотя примеры в этой главе иллюстрируют более сложную технику программирования, в них также подчеркнута поддержка в Python создания повторно используемого программного обеспечения. Реализации объектов, написанные с помощью классов, модулей и других средств Python, естественным образом становятся компонентами, полезными в целом, и могут быть использованы в любой импортирующей их программе. В сущности, мы будем создавать библиотеки классов структур данных, даже не планируя этого делать.

Кроме того, хотя примеры этой главы представляют собой чистый код Python, мы также проложим здесь дорогу к следующей части книги. С самой общей точки зрения новые объекты Python могут быть реализованы на Python или интегрированном языке, таком как C. В частности, обратите внимание на объекты стеков, реализуемые в первом разделе этой главы; позднее они будут заново реализованы на C, чтобы оценить как выгоды, так и сложность миграции на C.

Реализация стеков

Стеки являются распространенной и простой структурой данных, используемой в целом ряде приложений: обработке языков, поиске на графах и т. д. Вкратце, стек представляет собой совокупность объектов типа «последним пришел, первым ушел»: эле-

мент, добавленный в совокупность последним, всегда оказывается следующим удаляемым. Клиенты используют со стеками следующие действия:

- Проталкивание элементов на вершину стека (pushing)
- Выталкивание элементов с вершины стека (popping)

В зависимости от требований клиентов могут также иметься средства решения таких задач, как проверка пустоты стека, загрузка верхнего элемента без выталкивания его, обход элементов стека, проверка наличия некоторого элемента и т. д.

В Python для реализации стека часто достаточно простого списка: так как списки допускают редактирование по месту, можно добавлять или удалять элементы с начала (слева) или с конца (справа). В табл. 17.1 приведены различные встроенные операции, которые могут использоваться для реализации поведения типа стека в списках Python, в зависимости от того, является ли «вершина» стека первым или последним узлом списка. В этой таблице строка `'c'` является верхним элементом стека.

Таблица 17.1. Стеки в виде списков

Операция	Вершина в конце списка	Вершина в начале списка	Вершина в начале списка
New	`stack=['a','b','c ']`	`stack=['c','b','a ']`	`stack=['c','b','a ']`
Push	`stack.append('d')`	`stack.insert(0,'d')`	`stack[0:0] = ['d']`
Pop[a]	`x = stack[-1];` `del stack[-1]`	`x = stack[0];` `del stack[:1]`	`x = stack[0];` `stack[:1] = []`

[a] На самом деле в Python 1.5 появился метод списка `pop`, который можно использовать совместно с `append` для реализации стеков: вместо *push* пишем `list.append(value)`, вместо *pop* пишем `x=list.pop()`. Метод `pop` равносилен выборке с последующим удалением последнего элемента со смещением −1 (как в последней строке колонки 2 табл. 17.1). Однако поскольку списки представляют собой тип, а не класс, для реализации каких-то особых действий могут потребоваться приемы работы с классами стеков, описываемые в данной главе.

Такое приспособление списка действует, и относительно быстро. Но при этом основанные на стеках программы привязываются к избранному представлению стека: все операции со стеком будут жестко закодированы. Если позднее потребуется изменить представление стека или расширить его базовые операции, мы попались: придется исправлять все использующие стеки программы.

Например, чтобы добавить логику, которая контролирует количество операций, выполняемых программой со стеком, пришлось бы добавлять код рядом с каждой жестко закодированной операцией со стеком. В большой системе такая операция может оказаться нетривиальной. Как будет показано в следующей главе, если стеки окажутся узким местом для эффективности системы, можно перейти на стеки, реализованные на C. Как правило, жестко закодированные данные со встроенными структурами данных не всегда поддерживают возможные изменения в будущем так, как этого хотелось бы.

Модуль stack

Более удачным подходом может оказаться инкапсуляция реализаций стеков с помощью инструментов Python для повторного использования кода. Начнем с реализации стека в виде модуля, содержащего список Python вместе с функциями, действующими над ним (см. пример 17.1).

Пример 17.1. PP2E\Dstruct\Basic\stack1.py

```
stack = []                              # при первом импорте
```

```
error = 'stack1.error'                          # локальные исключения

def push(obj):
    global stack                                # 'global' для изменения
    stack = [obj] + stack                       # добавить элемент в начало

def pop():
    global stack
    if not stack:
        raise error, 'stack underflow'          # возбудить локальную ошибку
    top, stack = stack[0], stack[1:]            # удалить элемент из начала
    return top

def top():
    if not stack:                               # возбудить локальную ошибку
        raise error, 'stack underflow'          # или произойдет IndexError
    return stack[0]

def empty():      return not stack              # в стеке []?
def member(obj):  return obj in stack           # элемент в стеке?
def item(offset): return stack[offset]          # элемент по индексу
def length():     return len(stack)             # количество записей
def dump():       print '<Stack:%s>' % stack
```

Этот модуль создает объект списка (`stack`) и экспортирует функции для управления доступом к нему. Стек объявляется глобальным в функциях, которые его изменяют, но не в тех, которые только читают его. В модуле также объявлен объект ошибки (`error`), с помощью которого можно перехватывать исключительные ситуации, возбуждаемые локально в этом модуле. Некоторые ошибки стека являются встроенными исключительными ситуациями: метод `item` вызывает `IndexError` для индексов, нарушающих границы.

Большинство функций стека просто поручают выполнение операции встроенному списку, представляющему стек. В действительности модуль служит просто оболочкой, в которую заключен список Python. Но этот дополнительный слой логики интерфейса обеспечивает независимость клиентов от фактической реализации стека, поэтому в дальнейшем можно будет изменить стек, не оказывая влияния на его клиентов.

Как всегда, лучше всего разобраться в таком коде, посмотрев на него в деле. Вот интерактивный сеанс, иллюстрирующий интерфейсы модуля:

```
C:\...\PP2E\Dstruct\Basic>python
>>> import stack1
>>> stack1.push('spam')
>>> stack1.push(123)
>>> stack1.top()
123
>>> stack1.stack
[123, 'spam']
>>> stack1.pop()
123
>>> stack1.dump()
<Stack:['spam']>
>>> stack1.pop()
'spam'
>>> stack1.empty()
1
>>> for c in 'spam': stack1.push(c)
...
>>> while not stack1.empty():
```

```
...     print stack1.pop(),
...
m a p s
```

Другие операции аналогичны, но главное, на что здесь нужно обратить внимание, это то, что все операции со стеком являются *функциями* модуля. Например, можно совершить обход стека, но нужно использовать цикл в обратном направлении и функцию обращения по индексу (`item`). Ничто не мешает клиенту обращаться (и изменять) `stack1.stack` непосредственно, но при этом теряется весь смысл таких интерфейсов.

Класс stack

Наибольшим недостатком стека, основанного на модуле, является, видимо, поддержка только одного объекта стека. Все клиенты модуля `stack` фактически пользуются одним и тем же стеком. Иногда такая особенность нужна: стек может служить объектом совместно используемой несколькими модулями памяти. Но для реализации стека как настоящего типа данных необходимо использовать классы.

Для иллюстрации определим полнофункциональный *класс* стека. Класс `Stack`, показанный в примере 17.2, определяет новый тип данных с разнообразным поведением. Как и модуль, класс использует для хранения помещаемых на стек объектов список Python. Но на этот раз у каждого экземпляра есть собственный список. В классе определены как «реальные» методы, так и методы с особыми именами, реализующие стандартные операции типа. Комментарии в коде описывают специальные методы.

Пример 17.2. PP2E\Dstruct\Basic\stack2.py

```
error = 'stack2.error'                      # при импорте: локальное исключение

class Stack:
    def __init__(self, start=[]):           # self есть объект экземпляра
        self.stack = []                     # start есть любая последовательность: stack..
        for x in start: self.push(x)
        self.reverse()                      # отменить обращение порядка push
    def push(self, obj):                    # методы: как модуль + self
        self.stack = [obj] + self.stack     # вершина-начало списка
    def pop(self):
        if not self.stack: raise error, 'underflow'
        top, self.stack = self.stack[0], self.stack[1:]
        return top
    def top(self):
        if not self.stack: raise error, 'underflow'
        return self.stack[0]
    def empty(self):
        return not self.stack                       # instance.empty()

    # overloads
    def __repr__(self):
        return '[Stack:%s]' % self.stack            # вывод, обратные штрихи,..
    def __cmp__(self, other):
        return cmp(self.stack, other.stack)         # '==', '>, '<=', '!=',..
    def __len__(self):
        return len(self.stack)                      # len(instance), не instance
    def __add__(self, other):
        return Stack(self.stack + other.stack)      # instance1 + instance2
    def __mul__(self, reps):
        return Stack(self.stack * reps)             # instance * reps
    def __getitem__(self, offset):
```

```
        return self.stack[offset]                # intance[offset], in, for
    def __getslice__(self, low, high):
        return Stack(self.stack[low : high])     # instance[low:high]
    def __getattr__(self, name):
        return getattr(self.stack, name)         # instance.sort()/reverse()/..
```

Теперь отдельные экземпляры создаются при вызове класса `Stack` как функции. Во многих отношениях операции в классе `Stack` реализованы точно так же, как в модуле `stack` из примера 17.1. Но здесь доступ к стеку квалифицируется через `self`, объект экземпляра. У каждого экземпляра свой атрибут `stack`, который ссылается на собственный список экземпляра. Кроме того, стеки экземпляров создаются и инициализируются в методе конструктора `__init__`, а не при импорте модуля. Создадим несколько стеков и посмотрим, как все это действует на практике:

```
>>> from stack2 import Stack
>>> x = Stack()                    # создать объект стека, поместить в него данные
>>> x.push('spam')
>>> x.push(123)
>>> x                              # __repr__ выводит стек
[Stack:[123, 'spam']]

>>> y = Stack()                    # два разных объекта стека
>>> y.push(3.1415)                 # у них раздельное содержимое
>>> y.push(x.pop())
>>> x, y
([Stack:['spam']], [Stack:[123, 3.1415]])


>>> z = Stack()                    # третий отдельный объект стека
>>> for c in 'spam': z.push(c)
...
>>> while z: print z.pop(),        # __len__ проверяет истинность стека
...
m a p s

>>> z = x + y                      # __add__ обрабатывает оператор + для стека
>>> z                              # содержит три разных типа
[Stack:['spam', 123, 3.1415]]
>>> for item in z: print item,     # __getitem__ выполняет
...
spam 123 3.1415
```

Подобно спискам и словарям, `Stack` определяет методы и операторы для обработки экземпляров с помощью квалификации атрибутов и выражений. Кроме того, он определяет метод метакласса `__getattr__` для перехвата ссылок на атрибуты, не определенные в классе, и передачи их вложенному объекту списка (для поддержки методов списка: `sort`, `append`, `reverse` и т. д.). Многие операции модуля становятся операторами в классе. В табл. 17.2 показаны эквивалентные операции модуля и класса (колонки 1 и 2) и приводится метод класса, который выполняет каждую из них (колонка 3).

Таблица 17.2. Сравнение операций модуля/класса

Операции модуля	Операции класса	Метод класса
`module.empty()`	`not instance`	`__len__`
`module.member(x)`	`x in instance`	`__getitem__`
`module.item(i)`	`instance[I]`	`__getitem__`
`module.length()`	`len(instance)`	`__len__`

Таблица 17.2 (продолжение)

Операции модуля	Операции класса	Метод класса
`module.dump()`	`print instance`	`__repr__`
`range()` *`counter loops`*	`for x in instance`	`__getitem__`
`manual loop logic`	`instance + instance`	`__add__`
`module.stack.reverse()`	`instance.reverse()`	`__getattr__`
`module.push/pop/top`	`instance.push/pop/top`	`push/pop/top`

В сущности, классы позволяют расширять набор встроенных типов Python с помощью многократно используемых типов, реализуемых в модулях Python. Основанные на классах типы могут использоваться так же, как встроенные типы: в зависимости от того, какие методы операций в них определены, классы могут реализовывать числа, преобразования и последовательности и быть изменяемыми или нет. Основанные на классах типы могут также занимать положение, промежуточное между этими категориями.

Индивидуальная настройка: мониторы производительности

Мы уже убедились, что классы поддерживают множественные экземпляры и лучше интегрируются с объектной моделью Python благодаря определению методов операторов. Одной из других важных причин использования классов является возможность дальнейшего расширения и настройки. Реализуя стеки с помощью класса, можно в дальнейшем добавлять подклассы, точнее определяющие реализацию в соответствии с возникшими требованиями.

Допустим, например, что мы стали использовать класс `Stack` из примера 17.2, но столкнулись с проблемами производительности. Один из способов выявить узкие места состоит в том, чтобы оснастить структуры данных логикой, ведущей статистику использования, которую можно проанализировать после выполнения клиентских приложений. Так как `Stack` является классом, такую новую логику можно добавить в подклассе, не трогая исходный модуль стека (или его клиенты). Подкласс примера 17.3 расширяет `Stack`, вводя слежение за суммарной частотой операций push/pop и регистрируя максимальный размер каждого экземпляра.

Пример 17.3. PP2E\Dstruct\Basic\stacklog.py

```
from stack2 import Stack                                # расширяет импортируемый Stack

class StackLog(Stack):                                  # подсчет push/pop, максимального размера
    pushes = pops = 0                                   # разделяемые/статические члены класса
    def __init__(self, start=[]):                       # могут быть переменными класса
        self.maxlen = 0
        Stack.__init__(self, start)
    def push(self, object):
        Stack.push(self, object)                        # реально выполнить push
        StackLog.pushes = StackLog.pushes + 1           # суммарная статистика
        self.maxlen = max(self.maxlen, len(self))       # статистика для экземпляра
    def pop(self):
        StackLog.pops = StackLog.pops + 1               # суммарный счетчик
        return Stack.pop(self)                          # не 'self.pops': экземпляр
    def stats(self):
        return self.maxlen, self.pushes, self.pops      # получить счетчики экземпляра
```

Этот подкласс действует так же, как первоначальный Stack: в него просто добавлена логика контроля. С помощью нового метода stats получается кортеж со статистикой экземпляра:

```
>>> from stacklog import StackLog
>>> x = StackLog()
>>> y = StackLog()                          # создать два объекта стека
>>> for i in range(3): x.push(i)            # поместить в них объект
...
>>> for c in 'spam':   y.push(c)
...
>>> x, y                                    # выполнить унаследованный __repr__
([Stack:[2, 1, 0]], [Stack:['m', 'a', 'p', 's']])
>>> x.stats(), y.stats()
((3, 7, 0), (4, 7, 0))
>>>
>>> y.pop(), x.pop()
('m', 2)
>>> x.stats(), y.stats()                    # свой размер, все push, все pop
((3, 7, 2), (4, 7, 2))
```

Обратите внимание на использование атрибутов *класса* для регистраций суммарного количества помещений в стек и загрузки со стека и атрибутов *экземпляра* для максимальной длины каждого экземпляра. Придавая атрибуты разным объектам, можно расширять или сужать область их действия.

Оптимизация: стеки дерева кортежей

Одним из удобств помещения объектов в классы-оболочки является возможность изменения базовой реализации без нарушения работы остальной части программы. Например, в будущем можно осуществить оптимизацию с минимальным воздействием: интерфейс сохраняется, даже когда изменяется внутреннее устройство. Есть ряд способов реализации стеков, эффективность которых различна. До сих пор для обмена данными с нашими стеками использовались срезы и конкатенация. Это довольно неэффективно: обе операции создают копии заключенного в оболочку объекта списка. Для больших стеков такой способ будет отнимать много времени.

Одним из способов добиться ускорения является полное изменение базовой структуры данных. Например, помещаемые на стек объекты можно хранить в бинарном дереве кортежей: каждый элемент можно записать как пару (object, tree), где object – это элемент, помещенный на стек, а tree – это либо другая пара кортежа, задающая остальной стек, либо None для обозначения пустого стека. Стек элементов [1,2,3,4] внутренне будет храниться как дерево кортежей (1,(2,(3,(4,None)))).

Такое представление, основанное на кортежах, аналогично понятию «cons-cells» в семействе языков Lisp: объект слева – это car, а остальная часть дерева справа – это cdr. Так как мы добавляем или удаляем только верхний кортеж при помещении элемента в стек или снятии со стека, использование такой структуры позволяет избежать копирования целого стека. Для больших стеков преимущество может оказаться существенным. Эти идеи реализованы в следующем классе, приведенном в примере 17.4.

Пример 17.4. PP2E\Dstruct\Basic\stack3.py

```
class Stack:
    def __init__(self, start=[]):              # инициализация из любой последовательности
        self.stack = None                      # даже других (быстрых) стеков
```

```
        for i in range(-len(start), 0):
            self.push(start[-i - 1])          # протолкнуть в обратном порядке
    def push(self, node):                     # рост дерева 'вверх/влево'
        self.stack = node, self.stack         # новый корневой кортеж: (узел, дерево)
    def pop(self):
        node, self.stack = self.stack         # удалить корневой кортеж
        return node                           # TypeError, если пуст
    def empty(self):
        return not self.stack                 # 'None'?
    def __len__(self):                        # при: len, not
        len, tree = 0, self.stack
        while tree:
            len, tree = len+1, tree[1]        # пройти правые поддеревья
        return len
    def __getitem__(self, index):             # при: x[i], in, for
        len, tree = 0, self.stack
        while len < index and tree:           # обход/подсчет узлов nodes
            len, tree = len+1, tree[1]
        if tree:
            return tree[0]                    # IndexError при нарушении границ
        else: raise IndexError                # остановка 'in' и 'for'
    def __repr__(self): return '[FastStack:' + `self.stack` + ']'
```

Метод `__getitem__` этого класса обрабатывает индексирование, проверки `in` и итерации цикла `for`, как и прежде, но в этой версии нужно выполнить обход дерева, чтобы найти узел по индексу. Обратите внимание, что это не подкласс первоначального класса `Stack`. Так как здесь почти все операции реализованы по-другому, от наследования пользы мало. Но клиенты, использующие только общие для обоих классов операции, могут использовать их взаимозаменяемым образом – чтобы перейти на другую реализацию, нужно лишь импортировать класс стека из другого модуля. Вот сеанс использования этой версии стека; если придерживаться только проталкивания, снятия, индексирования и итерации, эта версия, по существу, неотличима от первоначальной:

```
>>> from stack3 import Stack
>>> x = Stack()
>>> y = Stack()
>>> for c in 'spam': x.push(c)
...
>>> for i in range(3): y.push(i)
...
>>> x
[FastStack:('m', ('a', ('p', ('s', None))))]
>>> y
[FastStack:(2, (1, (0, None)))]

>>> len(x), x[2], x[-1]
(4, 'p', 'm')
>>> x.pop()
'm'
>>> x
[FastStack:('a', ('p', ('s', None)))]
>>>
>>> while y: print y.pop(),
...
2 1 0
```

Оптимизация: модификация списка по месту

Однако для повышения скорости работы объекта стека лучше, наверное, вернуться к объекту списка Python и воспользоваться его изменяемостью. Поскольку списки допускают изменение по месту, их можно модифицировать быстрее, чем во всех предыдущих примерах. Операции изменения по месту типа `append` могут приводить к осложнениям, когда к списку обращаются из нескольких мест. Но так как список внутри объекта стека не предназначается для прямого доступа, то здесь, вероятно, мы защищены. Модуль примера 17.5 показывает один из способов реализации стека с изменениями по месту. Для простоты некоторые методы перегрузки операторов опущены. Новый метод Python `pop`, который здесь применен, эквивалентен доступу по индексу и удалению элемента со смещением –1 (здесь вершина находится в конце списка).

Пример 17.5. PP2E\Dstruct\Basic\stack4.py

```
error = 'stack4.error'                          # при импорте: локальное исключение

class Stack:
    def __init__(self, start=[]):               # self есть объект экземпляра
        self.stack = []                         # start есть любая последовательность: stack..
        for x in start: self.push(x)
    def push(self, obj):                        # методы: как модуль + self
        self.stack.append(obj)                  # вершина-конец списка
    def pop(self):
        if not self.stack: raise error, 'underflow'
        return self.stack.pop()                 # как выборка и удаление stack[-1]
    def top(self):
        if not self.stack: raise error, 'underflow'
        return self.stack[-1]
    def empty(self):
        return not self.stack                   # instance.empty()
    def __len__(self):
        return len(self.stack)                  # len(instance), не intance
    def __getitem__(self, offset):
        return self.stack[offset]               # instance[offset], in, for
    def __repr__(self): return '[Stack:%s]' % self.stack
```

Эта версия тоже работает, как оригинал в модуле `stack2` – просто замените `stack2` на `stack4` в предыдущем диалоге, и вы получите представление о его работе. Единственное заметное отличие в том, что элементы стека выводятся в обратном порядке (то есть вершина является концом):

```
>>> from stack4 import Stack
>>> x = Stack()
>>> x.push('spam')
>>> x.push(123)
>>> x
[Stack:['spam', 123]]
>>>
>>> y = Stack()
>>> y.push(3.1415)
>>> y.push(x.pop())
>>> x, y
([Stack:['spam']], [Stack:[3.1415, 123]])
>>> y.top()
123
```

Хронометраж усовершенствований

Объект стека с изменениями по месту работает, вероятно, быстрее, чем первоначальная версия и версия с деревом кортежей, но единственный способ действительно выяснить, насколько быстрее, – это провести хронометраж альтернативных реализаций. Поскольку такая операция может потребоваться неоднократно, определим сначала общий модуль для хронометража функций в Python. В примере 17.6 встроенный модуль `time` предоставляет функцию `clock`, с помощью которой можно получить текущее время CPU в виде секунд, выраженных числом с плавающей точкой, и функция `timer.test` просто вызывает нужную функцию `reps` раз и возвращает количество истекших секунд, вычитая время конца из времени начала.

Пример 17.6. PP2E\Dstruct\Basic\timer.py

```
def test(reps, func, *args):
    import time
    start = time.clock()            # текущее время CPU в секундах
    for i in xrange(reps):          # вызов func reps раз
        apply(func, args)           # отбросить возвращаемое значение
    return time.clock() - start     # время конца – время начала
```

Затем определим управляющий сценарий теста (см. пример 17.7). Он принимает три аргумента командной строки: количество операций push, pop и доступа по индексу (будем менять эти аргументы для проверки разных ситуаций). При выполнении на верхнем уровне сценарий создает 200 экземпляров исходного и оптимизированного классов стека и выполняет заданное количество операций с каждым стеком.[1] Операции push и pop изменяют стек; при индексировании происходит только выборка из него.

Пример 17.7. PP2E\Dstruct\Basic\stacktime.py

```
import stack2           # стеки на основе списков: [x]+y
import stack3           # стеки на дереве выборки: (x,y)
import stack4           # стеки изменения по месту:   y.append(x)
import timer            # общая функция таймера

rept = 200
from sys import argv
pushes, pops, items = eval(argv[1]), eval(argv[2]), eval(argv[3])

def stackops(stackClass):
    #print stackClass.__module__
    x = stackClass('spam')                  # создать объект стека
    for i in range(pushes): x.push(i)       # применить его методы
    for i in range(items):  t = x[i]
    for i in range(pops):   x.pop()

print 'stack2:', timer.test(rept, stackops, stack2.Stack)      # передать класс в тест
print 'stack3:', timer.test(rept, stackops, stack3.Stack)      # rept*(push+pop+ix)
print 'stack4:', timer.test(rept, stackops, stack4.Stack)
```

Вот некоторые данные, которые сообщил этот управляющий сценарий теста. Три вывода представляют измерения времени выполнения в секундах для исходного стека с кортежами и изменением по месту. Для каждого типа стека первый тест создает

[1] Если у вас под рукой есть первое издание этой книги, вы можете обратить внимание, что мне пришлось увеличить все коэффициенты теста, чтобы приблизиться к показателям времени выполнения, отмеченным раньше. И Python, и чипы за пять лет стали значительно быстрее.

200 объектов стека и выполняет примерно 120 000 операций со стеком (200 повторений × (200 push + 200 индексаций + 200 pop)) в течение приведенного времени выполнения теста. Результаты были получены на машине 650 МГц Pentium III под Windows с установленным Python 1.5.2:

```
C:\...\PP2E\Dstruct\Basic>python stacktime.py 200 200 200
stack2: 1.67890008213
stack3: 7.70020952413
stack4: 0.694291724635

C:\...\PP2E\Dstruct\Basic>python stacktime.py 200 50 200
stack2: 1.06876246669
stack3: 7.48964866994
stack4: 0.477584270605

C:\...\PP2E\Dstruct\Basic>python stacktime.py 200 200 50
stack2: 1.34536448817
stack3: 0.795615917129
stack4: 0.57297976835

C:\...\PP2E\Dstruct\Basic>python stacktime.py 200 200 0
stack2: 1.33500477715
stack3: 0.300776077373
stack4: 0.533050336077
```

Если внимательно посмотреть, то можно заметить по результатам, что стек, основанный на кортежах (`stack3`), показывает лучшую производительность, когда производится больше push и pop, но хуже работает, когда производится много обращений по индексу. Индексация происходит очень быстро для встроенных списков, но очень медленно для деревьев кортежей – класс Python должен обходить дерево вручную. Стеки с изменением по месту (`stack4`) почти всегда действуют быстрее всего, если только не делается вообще никакой индексации – в последнем тесте кортежи победили с небольшим перевесом. Поскольку push и pop – главное, что нужно клиентам от стека, кортежи являются претендентом на победу, несмотря на слабую производительность при доступе по индексу. Конечно, мы говорим о долях секунды после многих десятков тысяч операций; во многих приложениях пользователи могут не почувствовать разницы.

Реализация множеств

Другой часто используемой структурой данных является *множество* – совокупность объектов, поддерживающих следующие операции:

Пересечение

Создать новое множество, состоящее из общих элементов.

Объединение

Создать новое множество, состоящее из элементов, принадлежащих хотя бы одному операнду.

Принадлежность

Проверить, содержится ли элемент в множестве.

Есть и другие операции, в зависимости от предполагаемого использования. Множества оказываются удобными для работы с более абстрактными сочетаниями групп. Например, если есть группа инженеров и группа авторов, можно выбрать тех, кто занимается обоими видами деятельности, устроив пересечение двух множеств. В объеди-

нении таких множеств будут содержаться оба типа людей, но каждый из них будет включен только один раз.

Списки, кортежи и строки Python близки к понятию множества: оператор `in` проверяет принадлежность, `for` производит итерацию и т. д. Мы введем операции, не поддерживаемые непосредственно последовательностями Python. Идея состоит в *расширении* встроенных типов в соответствии с особыми требованиями.

Функции множеств

Как и прежде, начнем с основанного на функциях менеджера множеств. Но на этот раз вместо управления совместно используемым объектом множества в модуле определим функции, реализующие операции множеств над передаваемыми им последовательностями Python (см. пример 17.8).

Пример 17.8. PP2E\Dstruct\Basic\inter.py

```
def intersect(seq1, seq2):
    res = []                        # начать с пустого списка
    for x in seq1:                  # просмотр первой последовательности
        if x in seq2:
            res.append(x)           # вставить общие элементы в конец
    return res

def union(seq1, seq2):
    res = list(seq1)                # сделать копию seq1
    for x in seq2:                  # добавить новые элементы в seq2
        if not x in res:
            res.append(x)
    return res
```

Эти функции действуют над последовательностями любого типа – списками, строками, кортежами и другими объектами, удовлетворяющими протоколам последовательностей, предполагаемым этими функциями (циклы `for`, проверки принадлежности `in`). На самом деле их можно использовать даже со смешанными типами объектов: последние две команды следующего кода вычисляют пересечение и объединение списка и кортежа. Как обычно в Python, значение имеет *интерфейс* объекта, а не конкретный тип:

```
C:\...\PP2E\Dstruct\Basic>python
>>> from inter import *
>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2), union(s1, s2)
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])
>>> intersect([1,2,3], (1,4))
[1]
>>> union([1,2,3], (1,4))
[1, 2, 3, 4]
```

Обратите внимание, что результат здесь всегда является списком независимо от типов передаваемых последовательностей. Справиться с этим можно путем преобразования типов или используя класс, чтобы обойти проблему (что мы и сделаем чуть ниже). Но преобразования типов становятся неясными, если операнды имеют смешанные типы. В какой тип нужно преобразовывать?

Поддержка нескольких операндов

Если мы намерены использовать функции пересечения и объединения в качестве универсальных инструментов, то полезным расширением будет поддержка нескольких аргументов (то есть больше двух). Функции в примере 17.9 используют возможность задания в Python списков аргументов переменной длины, чтобы вычислять пересечение и объединение произвольного количества операндов.

Пример 17.9. PP2E\Dstruct\Basic\inter2.py

```
def intersect(*args):
    res = []
    for x in args[0]:                   # просмотр первого списка
        for other in args[1:]:          # для всех остальных аргументов
            if x not in other: break    # этот элемент везде?
        else:
            res.append(x)               # добавить общие элементы в конец

    return res
def union(*args):
    res = []
    for seq in args:                    # для всех аргументов-последовательностей
        for x in seq:                   # для всех узлов в аргументе
            if not x in res:
                res.append(x)           # добавить новые элементы в результат
    return res
```

Функции с несколькими операндами действуют над последовательностями так же, как исходные функции, но поддерживают три и более операндов. Обратите внимание, что последние два примера в следующем сеансе действуют со списками, содержащими составные аргументы: проверки `in` в функциях пересечения и объединения для определения результатов сравнения с совокупностью рекурсивно применяют проверку равенства к узлам последовательностей на необходимую глубину:

```
C:\...\PP2E\Dstruct\Basic>python
>>> from inter2 import *
>>> s1, s2, s3 = 'SPAM', 'SLAM', 'SCAM'
>>> intersect(s1, s2)
['S', 'A', 'M']
>>> intersect(s1, s2, s3)
['S', 'A', 'M']
>>> intersect(s1, s2, s3, 'HAM')
['A', 'M']

>>> union(s1, s2), union(s1, s2, s3)
(['S', 'P', 'A', 'M', 'L'], ['S', 'P', 'A', 'M', 'L', 'C'])
>>> intersect([1,2,3], (1,4), range(5))
[1]

>>> s1 = (9, (3.14, 1), "bye", [1,2], "mello")
>>> s2 = [[1,2], "hello", (3.14, 0), 9]
>>> intersect(s1, s2)
[9, [1, 2]]
>>> union(s1, s2)
[9, (3.14, 1), 'bye', [1, 2], 'mello', 'hello', (3.14, 0)]
```

Классы множеств

Функции множеств могут действовать над различными последовательностями, но они не столь дружественны, как подлинные объекты. Помимо всего прочего, сценариям может потребоваться вручную следить за последовательностями, передаваемыми в эти функции. Классы могут действовать успешнее: класс примера 17.10 реализует объект множества, который может содержать объекты любого типа. Подобно классам стеков, они в сущности являются оболочками вокруг списков Python с дополнительными операциями множеств.

Пример 17.10. PP2E\Dstruct\Basic\set.py

```
class Set:
    def __init__(self, value = []):     # при создании объекта
        self.data = []                   # управляет локальным списком
        self.concat(value)
    def intersect(self, other):          # other - последовательность любого типа
        res = []                         # self - объект экземпляра
        for x in self.data:
            if x in other:
                res.append(x)
        return Set(res)                  # возвратить новое множество
    def union(self, other):
        res = self.data[:]               # сделать копию собственного списка
        for x in other:
            if not x in res:
                res.append(x)
        return Set(res)
    def concat(self, value):             # value: список, строка, множество...
        for x in value:                  # отфильтровывает дубликаты
           if not x in self.data:
                self.data.append(x)

    def __len__(self):          return len(self.data)
    def __getitem__(self, key): return self.data[key]
    def __and__(self, other):   return self.intersect(other)
    def __or__(self, other):    return self.union(other)
    def __repr__(self):         return '<Set:' + `self.data` + '>'
```

Класс `Set` используется так же, как класс `Stack`, рассмотренный выше в этой главе: мы создаем экземпляры и применяем к ним операторы последовательностей плюс особые операции множеств. Пересечение и объединение можно вызывать как методы или с помощью операторов `&` и `|`, обычно используемых со встроенными целочисленными объектами. Поскольку теперь в выражениях можно записывать операторы в строку (например, `x & y & z`), очевидно, нет нужды поддерживать здесь несколько операндов в методах пересечения/объединения. Как и для всех объектов, можно использовать класс `Set` внутри программы или тестировать его интерактивно следующим образом:

```
>>> from set import Set
>>> users1 = Set(['Bob', 'Emily', 'Howard', 'Peeper'])
>>> users2 = Set(['Jerry', 'Howard', 'Carol'])
>>> users3 = Set(['Emily', 'Carol'])
>>> users1 & users2
<Set:['Howard']>
>>> users1 | users2
<Set:['Bob', 'Emily', 'Howard', 'Peeper', 'Jerry', 'Carol']>
>>> users1 | users2 & users3
<Set:['Bob', 'Emily', 'Howard', 'Peeper', 'Carol']>
>>> (users1 | users2) & users3
```

```
<Set:['Emily', 'Carol']>
>>> users1.data
['Bob', 'Emily', 'Howard', 'Peeper']
```

Оптимизация множеств: переход на словари

Первой проблемой, с которой можно столкнуться, начав использовать класс `Set`, оказывается производительность: его вложенные циклы `for` и просмотры `in` замедляются с экспоненциальной скоростью. Такая медлительность иногда оказывается существенной в приложениях, но библиотечные классы должны кодироваться с максимально возможной эффективностью.

Один из способов оптимизировать производительность множеств состоит в переходе в реализации к словарям вместо списков для внутреннего хранения множеств – элементы могут храниться в виде ключей словаря, все значения в котором равны `None`. Так как время поиска в словарях постоянно и невелико, просмотр списка `in` в первоначальном множестве в этой схеме можно заменить прямой выборкой из словаря. На традиционном языке переход в множествах на словари заменяет медленный линейный поиск быстрыми хеш-таблицами.

Модуль в примере 17.11 реализует эту идею. Его класс является подклассом исходного множества и переопределяет методы, связанные с внутренним представлением, но наследует остальные. Унаследованные методы `&` и `|` запускают здесь новые методы пересечения и объединения, а унаследованный метод `len` действует со словарями без изменений. Поскольку клиенты `Set` не зависят от порядка элементов в множестве, они могут непосредственно переключиться на эту версию, просто изменив имя модуля, из которого импортируется `Set`; имя класса остается прежним.

Пример 17.11. PP2E\Dstruct\Basic\fastset.py

```
import set
                                        # fastset.Set расширяет set.Set
class Set(set.Set):
    def __init__(self, value = []):
        self.data = {}                  # управляет локальным словарем
        self.concat(value)              # хеширование: время поиска изменяется линейно
    def intersect(self, other):
        res = {}
        for x in other:                 # other: последовательность или Set
            if self.data.has_key(x):    # поиск с помощью хеш-таблицы
                res[x] = None
        return Set(res.keys())          # новый Set на базе словаря
    def union(self, other):
        res = {}                        # other: последовательность или Set
        for x in other:                 # сканировать каждое множество один раз
            res[x] = None
        for x in self.data.keys():      # '&' и '|' здесь возвращаются
            res[x] = None               # так получаются новые «быстрые» множества
        return Set(res.keys())
    def concat(self, value):
        for x in value: self.data[x] = None

    # наследование and, or, len
    def __getitem__(self, key):  return self.data.keys()[key]
    def __repr__(self):          return '<Set:' + `self.data.keys()` + '>'
```

Эта версия работает примерно так же, как предыдущая:

```
>>> from fastset import Set
```

```
>>> users1 = Set(['Bob', 'Emily', 'Howard', 'Peeper'])
>>> users2 = Set(['Jerry', 'Howard', 'Carol'])
>>> users3 = Set(['Emily', 'Carol'])
>>> users1 & users2
<Set:['Howard']>
>>> users1 | users2
<Set:['Emily', 'Howard', 'Jerry', 'Carol', 'Peeper', 'Bob']>
>>> users1 | users2 & users3
<Set:['Emily', 'Howard', 'Carol', 'Peeper', 'Bob']>
>>> (users1 | users2) & users3
<Set:['Emily', 'Carol']>
>>> users1.data
{'Emily': None, 'Bob': None, 'Peeper': None, 'Howard': None}
```

Главным функциональным отличием этой версии является *порядок* элементов в множестве: поскольку словари упорядочиваются произвольным образом, порядок будет отличаться от исходного. Например, можно хранить в множествах составные объекты, но порядок элементов в этой версии отличается:

```
>>> import set, fastset
>>> a = set.Set([(1,2), (3,4), (5,6)])
>>> b = set.Set([(3,4), (7,8)])
>>> a & b
<Set:[(3, 4)]>
>>> a | b
<Set:[(1, 2), (3, 4), (5, 6), (7, 8)]>

>>> a = fastset.Set([(1,2), (3,4), (5,6)])
>>> b = fastset.Set([(3,4), (7,8)])
>>> a & b
<Set:[(3, 4)]>
>>> a | b
<Set:[(3, 4), (1, 2), (7, 8), (5, 6)]>
>>> b | a
<Set:[(3, 4), (5, 6), (1, 2), (7, 8)]>
```

Во всяком случае от множеств не требуется упорядоченность, поэтому здесь нет накладки. А вот отклонение, которое может иметь значение, состоит в том, что в этой версии нельзя хранить нехешируемые объекты. Это следует из того, что ключи словаря должны быть неизменяемыми. Так как значения хранятся в ключах, множества, основанные на словарях, могут содержать только такие элементы, как кортежи, строки, числа и объекты классов с неизменными сигнатурами. Изменяемые объекты типа списков и словарей использовать непосредственно нельзя. Например, вызов:

```
fastset.Set([[1,2],[3,4]])
```

возбуждает исключительную ситуацию для этого множества на базе словарей, но действует с исходным классом множеств. Кортежи работают здесь потому, что они неизменяемы; Python вычисляет хеш-значения и проверяет равенство ключей, как ожидалось.

Хронометраж работы

А как обстоят дела с оптимизацией? Пример 17.12 содержит сценарий, сравнивающий производительность классов множеств. Он повторно использует модуль `timer`, с помощью которого мы ранее тестировали стеки.

Пример 17.12. PP2E\Dstruct\Basic\settime.py

```
import timer, sys
import set, fastset
```

```
def setops(Class):
    a = Class(range(50))                # множество из 50 целых
    b = Class(range(20))                # множество из 20 целых
    c = Class(range(10))
    d = Class(range(5))
    for i in range(5):
        t = a & b & c & d               # 3 пересечения
        t = a | b | c | d               # 3 объединения

if __name__ == '__main__':
    rept = int(sys.argv[1])
    print 'set =>    ', timer.test(rept, setops, set.Set)
    print 'fastset =>', timer.test(rept, setops, fastset.Set)
```

Функция `setops` создает четыре множества и пять раз обрабатывает их операторами пересечения и объединения. Аргумент командной строки задает число повторений всего процесса. Точнее, при каждом вызове `setops` создается 34 экземпляра `Set` ($4 + [5 \times (3 + 3)]$) и выполняются методы `intersect` и `union` по 15 раз каждый (5×3) в теле цикла `for`. На той же самой тестовой машине и на этот раз достигается резкий рост производительности:

```
C:\...\PP2E\Dstruct\Basic>python settime.py 50
set =>     1.5440352671
fastset => 0.446057593993

C:\...\PP2E\Dstruct\Basic>python settime.py 100
set =>     2.77783486146
fastset => 0.888354648921

C:\...\PP2E\Dstruct\Basic>python settime.py 200
set =>     5.7762634305
fastset => 1.77677885985
```

По крайней мере, в этом контрольном примере простая реализация множества втрое медленнее, чем для множеств, основанных на словарях. На самом деле такое троекратное ускорение вполне обоснованно. Словари Python уже являются оптимизированными хеш-таблицами, усовершенствовать их весьма затруднительно. Пока не появится свидетельств того, что основанные на словарях таблицы все же слишком медлительны, наша работа здесь закончена.

Программные приемы оптимизации fastset

В существующем виде класс `fastset` содержит узкое место: при каждом вызове метода словаря `keys` Python создает новый список для хранения результата, и это может происходить многократно при получении пересечений и объединений. Если вы хотите еще больше оптимизировать этот класс, посмотрите следующие файлы на прилагаемом к книге CD:

- *PP2E\Dstruct\Basic\fastset2.py*
- *PP2E\Dstruct\Basic\fastset3.py*

Я написал их в попытке еще более ускорить работу множеств, но потерпел поражение. Оказывается, добавляемый с целью сокращения операций код обычно сводит на нет достигаемое ускорение. Возможно, есть более быстрый код, но наибольший выигрыш здесь, вероятно, был достигнут изменением базовой структуры данных на словари, а не мелкими поправками в коде.

Практика показывает, что интуитивные представления о производительности почти всегда оказываются ошибочными в таких динамических языках, как Python: обычно

Использование профайлера Python

Профайлер Python дает еще один способ собрать данные по производительности помимо кода для хронометража, используемого в этой главе. Поскольку профайлер прослеживает все вызовы функций, он позволяет сразу получить значительно больше информации. Благодаря этому он предоставляет более мощный путь выявления узких мест в медленных программах – после профилирования можно составить хорошее представление о том, где должны быть сосредоточены усилия по оптимизации.

Профайлер поступает вместе с Python как стандартный библиотечный модуль `profile`, который предоставляет ряд интерфейсов для измерения производительности кода. Он устроен и используется во многом аналогично отладчику командной строки `pdb`: для измерения производительности импортируйте модуль `profile` и вызывайте его функции со строкой кода. Простейшим интерфейсом профилирования служит функция `profile.run(statementstring)`. При вызове профайлер выполняет строку кода, собирает во время выполнения статистику и по выполнении операции выдает на экран отчет.

Формат отчета прост и хорошо документирован в библиотечном руководстве Python. По умолчанию в нем перечисляются количество вызовов и время, затраченное каждой вызываемой во время прогона функцией. При включении профайлера каждое событие интерпретатора маршрутизируется обработчику Python. При этом получается точная картина производительности, но профилируемая программа выполняется значительно медленнее, чем обычно.

виновником проблем производительности оказывается алгоритм, а не стиль программирования и даже не язык реализации. После удаления комбинаторного алгоритма просмотра списка из первоначального класса множества реализация Python стала значительно быстрее.

В действительности перевод исходного класса множества на C без исправления алгоритма не решает реальной проблемы производительности. Программные уловки тоже обычно не сильно помогают, зато делают программы труднопонимаемыми. В Python почти всегда лучше сначала писать программу так, чтобы ее можно было легко прочесть, а затем уже при необходимости оптимизировать. Несмотря на свою простоту, `fastset` действительно работает быстро.

Алгебра отношений для множеств (CD)

Если вас интересует кодирование на Python дополнительных операций с множествами, посмотрите следующие файлы на прилагаемом к книге CD:

- *PP2E\Dstruct\Basic\rset.py*: реализация `RSet`
- *PP2E\Dstruct\Basic\reltest.py*: тестовый сценарий для `RSet`

Подкласс `Rset`, определяемый в *rset.py*, вводит основные операции алгебры отношений в множества, основанные на словарях. В нем предполагается, что элементы множеств являются отображениями (рядами) с одним элементом в колонке (поле). `RSet` наследует все исходные операции `Set` (итерацию, пересечение, объединение, операторы `&` и `|`, фильтрацию дубликатов и прочее) и добавляет новые операции в виде методов:

Select

Возвращает множество узлов, в которых поле равно заданному значению.

Bagof

Собирает узлы, удовлетворяющие строке выражения.

Find

Отбирает кортежи согласно сравнению, полю и значению.

Match

Находит в двух множествах узлы с одинаковыми значениями в общих полях.

Product

Вычисляет Декартово произведение: соединение кортежей из двух множеств.

Join

Объединяет кортежи из двух множеств, в которых поле имеет одинаковое значение.

Project

Извлекает указанные поля из кортежей в таблице.

Difference

Удаляет кортежи одного множества из другого.

Альтернативные реализации *разности* множеств есть в файле *diff.py* в том же каталоге на CD.

Двоичные деревья поиска

Двоичные деревья представляют собой структуры данных, упорядочивающие вставляемые узлы: элементы, меньшие узла, записываются в его левое поддерево, а элементы, большие узла, помещаются в правое. Самые нижние поддеревья пусты. Благодаря такой структуре двоичные деревья естественным образом поддерживают быстрый рекурсивный обход – по крайней мере, в идеале при каждом переходе к поддереву пространство поиска сокращается вдвое.[1]

Двоичные деревья названы так из-за подразумеваемой древовидной структуры ссылок на поддеревья. Обычно их узлы реализуются как тройки значений: `(ЛевоеПоддерево, ЗначениеУзла, ПравоеПоддерево)`. За пределами этого в реализации деревьев существует довольно широкая свобода. Здесь мы будем использовать подход на основе классов:

- *BinaryTree* служит главным объектом, который инициализирует и контролирует фактическое дерево.
- *EmptyNode* является пустым объектом, совместно используемым всеми пустыми поддеревьями (внизу).
- *BinaryNode* – объекты, представляющие непустые узлы дерева, имеющие значение и два поддерева.

Чтобы не писать отдельные функции поиска, двоичные деревья строятся из «умных» объектов (экземпляров классов), которые умеют обрабатывать запросы вставки/поиска и вывода и передавать их объектам поддеревьев. Фактически это еще один пример действия композиционной связи ООП: в одни узлы дерева вкладываются другие и запросы на поиск передаются вложенным поддеревьям. Единственный экземпляр пустого класса совместно используется всеми пустыми поддеревьями двоичного дерева, а при вставке `EmptyNode` заменяется внизу на `BinaryNode` (см. пример 17.13).

Пример 17.13. PP2E\Dstruct\Classics\btree.py

```
class BinaryTree:
```

[1] Если вы ищете более графическое изображение двоичных деревьев, перейдите к примеру PyTree в конце главы или просто запустите его на своей машине.

```
    def __init__(self):       self.tree = EmptyNode()
    def __repr__(self):       return `self.tree`
    def lookup(self, value):  return self.tree.lookup(value)
    def insert(self, value):  self.tree = self.tree.insert(value)

class EmptyNode:
    def __repr__(self):
        return '*'
    def lookup(self, value):                              # отказ внизу
        return 0
    def insert(self, value):
        return BinaryNode(self, value, self)              # добавить вниз новый узел

class BinaryNode:
    def __init__(self, left, value, right):
        self.data, self.left, self.right  =  value, left, right
    def lookup(self, value):
        if self.data == value:
            return 1
        elif self.data > value:
            return self.left.lookup(value)                # смотреть слева
        else:
            return self.right.lookup(value)               # смотреть справа
    def insert(self, value):
        if self.data > value:
            self.left = self.left.insert(value)           # рост влево
        elif self.data < value:
            self.right = self.right.insert(value)         # рост вправо
        return self
    def __repr__(self):
        return '( %s, %s, %s )' % (`self.left`, `self.data`, `self.right`)
```

Как обычно, `BinaryTree` может содержать объекты любого типа, поддерживающего предполагаемый протокол интерфейса – в данном случае сравнения `>` и `<`. В их число входят экземпляры классов с методом `__cmp__`. Поэкспериментируем с интерфейсами этого модуля. В следующем коде в новое дерево заносятся пять целых чисел, а затем ищутся значения 0...9:

```
C:\...\PP2E\Dstruct\Classics>python
>>> from btree import BinaryTree
>>> x = BinaryTree()
>>> for i in [3,1,9,2,7]:  x.insert(i)
...
>>> for i in range(10):  print (i, x.lookup(i)),
...
(0, 0) (1, 1) (2, 1) (3, 1) (4, 0) (5, 0) (6, 0) (7, 1) (8, 0) (9, 1)
```

Чтобы посмотреть, как растет это дерево, добавьте команду print после каждой вставки. Узлы деревьев выводят себя как триплеты, а пустые узлы – как `*`. В результате отражена вложенность деревьев:

```
>>> y = BinaryTree()
>>> y
*
>>> for i in [3,1,9,2,7]:
...     y.insert(i); print y
...
( *, 3, * )
( ( *, 1, * ), 3, * )
```

```
( ( *, 1, * ), 3, ( *, 9, * ) )
( ( *, 1, ( *, 2, * ) ), 3, ( *, 9, * ) )
( ( *, 1, ( *, 2, * ) ), 3, ( ( *, 7, * ), 9, * ) )
```

В конце этой главы мы увидим другой способ визуализации деревьев в GUI (что означает приглашение перелистать сейчас книгу до того места). Значениями узлов в этом объекте дерева могут быть любые допускающие сравнение объекты Python; вот, например, деревья строк:

```
>>> z = BinaryTree()
>>> for c in 'badce':  z.insert(c)
...
>>> z
( ( *, 'a', * ), 'b', ( ( *, 'c', * ), 'd', ( *, 'e', * ) ) )
>>> z = BinaryTree()
>>> for c in 'abcde': z.insert(c)
...
>>> z
( *, 'a', ( *, 'b', ( *, 'c', ( *, 'd', ( *, 'e', * ) ) ) ) )
```

Обратите здесь внимание на последний результат: если вставляемые в двоичное дерево элементы уже упорядочены, то получается *линейная* структура и утрачивается механизм деления пространства поиска двоичных деревьев (у дерева растут только правые ветви). Это наихудшая ситуация, и на практике двоичные деревья хорошо поступают, разделяя значения. Но если вы хотите заняться этой темой дальше, почитайте книгу по структурам данных, в которой описывается технология балансировки деревьев для обеспечения их максимальной густоты.

Заметьте также, что для простоты эти деревья хранят только значения, а поиск возвращает 1 или 0 («истина» или «ложь»). На практике иногда требуется хранить вместе ключ и ассоциируемое значение (и даже больше) в каждом узле дерева. В примере 17.14 для лесорубов, которые могут оказаться среди читателей, показано, как выглядит такой объект дерева.

Пример 17.14. PP2E\Dstruct\Classics\btree-keys.py

```
class KeyedBinaryTree:
    def __init__(self):          self.tree = EmptyNode()
    def __repr__(self):          return `self.tree`
    def lookup(self, key):       return self.tree.lookup(key)
    def insert(self, key, val):  self.tree = self.tree.insert(key, val)

class EmptyNode:
    def __repr__(self):
        return '*'
    def lookup(self, key):                                  # отказ внизу
        return None
    def insert(self, key, val):
        return BinaryNode(self, key, val, self)             # добавить узел вниз

class BinaryNode:
    def __init__(self, left, key, val, right):
        self.key,  self.val   = key, val
        self.left, self.right = left, right
    def lookup(self, key):
        if self.key == key:
            return self.val
        elif self.key > key:
            return self.left.lookup(key)                    # смотреть слева
```

```
            else:
                return self.right.lookup(key)                # смотреть справа
        def insert(self, key, val):
            if self.key == key:
                self.val = val
            elif self.key > key:
                self.left = self.left.insert(key, val)       # рост влево
            elif self.key < key:
                self.right = self.right.insert(key, val)     # рост вправо
            return self
        def __repr__(self):
            return (`( %s, %s=%s, %s )` % (`self.left`, `self.key`, `self.val`, `self.right`))

    if __name__ == '__main__':
        t = KeyedBinaryTree()
        for (key, val) in [('bbb', 1), ('aaa', 2), ('ccc', 3)]:
            t.insert(key, val)
        print t
        print t.lookup('aaa'), t.lookup('ccc')
        t.insert('ddd', 4)
        t.insert('aaa', 5)                                   # изменяет значение ключа
        print t
```

А вот результат работы кода самотестирования этого сценария; на этот раз в узлах просто больше содержимого:

```
C:\...\PP2E\Dstruct\Classics>python btree-keys.py
( ( *, 'aaa'=2, * ), 'bbb'=1, ( *, 'ccc'=3, * ) )
2 3
( ( *, 'aaa'=5, * ), 'bbb'=1, ( *, 'ccc'=3, ( *, 'ddd'=4, * ) ) )
```

Поиск на графах

Многие задачи представляются в виде графа, который является множеством состояний с переходами («дугами»), ведущими из одного состояния в другое. Например, планирование маршрута путешествия в действительности является замаскированной задачей поиска на графе – состояния представляют те места, где вы хотите побывать, а дуги представляют транспортные маршруты между ними.

В этом разделе представлены простые программы Python, осуществляющие поиск в направленном циклическом графе путей между начальным состоянием и конечной точкой. Графы являются более общими конструкциями, чем деревья, потому что ссылки смогут указывать на произвольные узлы – даже на уже посещавшиеся (отсюда название «циклический»).

Граф, используемый для тестирования программ поиска этого раздела, изображен на рис. 17.1. Стрелки на концах дуг указывают разрешенные маршруты (например, из *A* можно попасть в *B*, *E* и *G*). Алгоритмы поиска обходят этот граф методом поиска в глубину и не допускают возврата в ту же точку, чтобы избежать зацикливания. Если представить себе, что это карта, на которой узлы представляют города, а дуги представляют дороги, то этот пример может показаться чуть более осмысленным.

Первое, что нужно сделать, это выбрать способ представления данного графа в сценарии Python. Один из подходов заключается в использовании встроенных типов данных и функций поиска. Файл примера 17.15 строит контрольный граф просто как словарь: каждое состояние является ключом словаря со списком ключей узлов, в которые можно из него попасть (то есть его дуг). В этом файле также определена функция, с помощью которой мы выполним несколько поисков на графе.

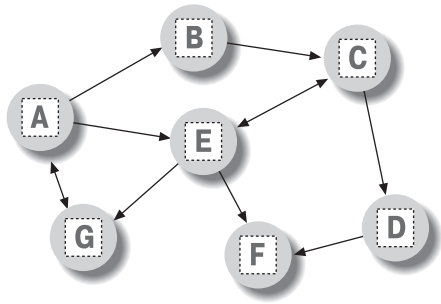


Рис. 17.1. Направленный граф

Пример 17.15. PP2E\Dstruct\Classics\gtestfunc.py

```
Graph = {'A':  ['B', 'E', 'G'],
         'B':  ['C'],                          # направленный циклический граф
         'C':  ['D', 'E'],                     # хранится в виде словаря
         'D':  ['F'],                          # 'ключ' ведет к [узлы]
         'E':  ['C', 'F', 'G'],
         'F':  [ ],
         'G':  ['A']  }

def tests(searcher):                           # проверка функции поиска
    print searcher('E', 'D', Graph)            # найти все пути из 'E' в 'D'
    for x in ['AG', 'GF', 'BA', 'DA']:
        print x, searcher(x[0], x[1], Graph)
```

Теперь напишем два модуля, реализующие реальные алгоритмы поиска. Оба они не зависят от графа, в котором производится поиск (он передается им в качестве аргумента). Первая функция поиска в примере 17.16 использует *рекурсию* для прохода по графу.

Пример 17.16. PP2E\Dstruct\Classics\gsearch1.py

```
# найти на графе все пути из начальной точки в конечную

def search(start, goal, graph):
    solns = []
    generate([start], goal, solns, graph)           # собрать пути
    solns.sort( lambda x, y: cmp(len(x), len(y)) )  # сортировать по длине пути
    return solns

def generate(path, goal, solns, graph):
    state = path[-1]
    if state == goal^                               # найти конечную точку
        solns.append(path)                          # изменить solns по месту
    else:                                           # проверить все дуги
        for arc in graph[state]:                    # выбросить циклы
            if arc not in path:
                generate(path + [arc], goal, solns, graph)

if __name__ == '__main__':
    import gtestfunc
    gtestfunc.tests(search)
```

Вторая функция поиска из примера 17.17 использует явный *стек*, который должен быть продолжен, и представление стека в виде дерева кортежей, рассматривавшееся выше в этой главе.

Пример 17.17. PP2E\Dstruct\Classics\gsearch2.py

```
# Использование стека путей вместо рекурсии

def search(start, goal, graph):
    solns = generate( ([start], []), goal, graph)
    solns.sort( lambda x, y: cmp(len(x), len(y)) )
    return solns

def generate(paths, goal, graph):               # возвращает список solns
    solns = []                                  # использовать стек кортежей
    while paths:
        front, paths = paths                    # вытолкнуть верхний путь
        state = front[-1]
        if state == goal:
            solns.append(front)                 # цель в этом пути
        else:
            for arc in graph[state]:            # добавить все продолжения
                if arc not in front:
                    paths = (front + [arc]), paths
    return solns

if __name__ == '__main__':
    import gtestfunc
    gtestfunc.tests(search)
```

Обе функции следят за узлами, посещенными вдоль пути, чтобы избежать циклов. Если продолжение оказывается в текущем пути, то возникает цикл. Полученный список решений сортируется по увеличению длины с помощью метода списка `sort` и встроенной функции сравнения `cmp`. Чтобы проверить модули поиска, просто запустите их; код самотестирования вызывает готовый тест поиска в модуле `gtestfunc`:

```
C:\...\PP2E\Dstruct\Classics>python gsearch1.py
[['E', 'C', 'D'], ['E', 'G', 'A', 'B', 'C', 'D']]
AG [['A', 'G'], ['A', 'E', 'G'], ['A', 'B', 'C', 'E', 'G']]
GF [['G', 'A', 'E', 'F'], ['G', 'A', 'B', 'C', 'D', 'F'],
    ['G', 'A', 'B', 'C', 'E', 'F'], ['G', 'A', 'E', 'C', 'D', 'F']]
BA [['B', 'C', 'E', 'G', 'A']]
DA []

C:\...\PP2E\Dstruct\Classics>python gsearch2.py
[['E', 'C', 'D'], ['E', 'G', 'A', 'B', 'C', 'D']]
AG [['A', 'G'], ['A', 'E', 'G'], ['A', 'B', 'C', 'E', 'G']]
GF [['G', 'A', 'E', 'F'], ['G', 'A', 'E', 'C', 'D', 'F'],
    ['G', 'A', 'B', 'C', 'E', 'F'], ['G', 'A', 'B', 'C', 'D', 'F']]
BA [['B', 'C', 'E', 'G', 'A']]
DA []
```

Этот вывод показывает списки возможных путей через контрольный граф; я добавил два переноса строки, чтобы облегчить его чтение. Обратите внимание, что обе функции во всех тестах находят одинаковые пути, но порядок нахождения может быть различным. Порядок `gsearch2` зависит от того, как и когда продолжения добавляются в его стек пути.

Перевод графов на классы

Использование словарей для представления графов эффективно: смежные узлы находятся быстрой операцией хеширования. Но для некоторых приложений более оправданными могут оказаться другие представления. Например, для моделирования уз-

лов в сети могут также использоваться классы, подобно двоичным деревьям в более раннем примере. В качестве классов узлы могут иметь содержание, полезное при более сложном поиске. Для иллюстрации в примере 17.18 показан альтернативный код функции поиска на графе; его алгоритм ближе всего к gsearch1.

Пример 17.18. PP2E\Dstruct\Classics\graph.py

```
# Построить граф из объектов, умеющих осуществлять поиск

class Graph:
    def __init__(self, label, extra=None):
        self.name = label                   # узлы=объекты экземпляров
        self.data = extra                   # граф=связанные объекты
        self.arcs = []
    def __repr__(self):
        return self.name
    def search(self, goal):
        Graph.solns = []
        self.generate([self], goal)
        Graph.solns.sort(lambda x,y: cmp(len(x), len(y)))
        return Graph.solns
    def generate(self, path, goal):
        if self == goal:                    # класс == цель
            Graph.solns.append(path)        # или self.solns: то же
        else:
            for arc in self.arcs:
                if arc not in path:
                    arc.generate(path + [arc], goal)

if __name__ == '__main__':
    import gtestobj1
    gtestobj1.tests(Graph)
```

В этой версии графы представляются как сеть вложенных объектов экземпляров класса. Каждый узел в графе содержит список объектов узлов, к которым он ведет (arcs) и в котором он умеет выполнять поиск. Метод generate проходит через объекты в графе. Но на этот раз переходы прямо доступны в списке arcs каждого узла; нет необходимости индексировать (или передавать) словарь, чтобы найти связанные объекты.

Для проверки модуль примера 17.19 снова строит контрольный граф, на этот раз с помощью связанных экземпляров класса Graph. Обратите внимание на использование exec в коде самотестирования: он выполняет динамически создаваемые строки, чтобы осуществить семь присваиваний (A=Graph('A'), B=Graph('B') и т. д.).

Пример 17.19. PP2E\Dstruct\Classics\gtestobj1.py

```
def tests(Graph):
    for name in "ABCDEFG":                          # сначала создать объекты
        exec "%s = Graph('%s')" % (name, name)      # label=variable-name

    A.arcs = [B, E, G]
    B.arcs = [C]                                    # теперь настроить их связи:
    C.arcs = [D, E]                                 # вложенный список экземпляров классов
    D.arcs = [F]
    E.arcs = [C, F, G]
    G.arcs = [A]

    A.search(G)
    for (start, stop) in [(E,D), (A,G), (G,F), (B,A), (D,A)]:
        print start.search(stop)
```

Этот тест выполняется путем запуска модуля graph, который передает класс графа, например:

```
C:\...\PP2E\Dstruct\Classics>python graph.py
[[E, C, D], [E, G, A, B, C, D]]
[[A, G], [A, E, G], [A, B, C, E, G]]
[[G, A, E, F], [G, A, B, C, D, F], [G, A, B, C, E, F], [G, A, E, C, D, F]]
[[B, C, E, G, A]]
[]
```

Результат такой же, как для функций, но имена узлов без кавычек: узлы в списках путей теперь являются экземплярами Graph, а схема класса __repr__ подавляет кавычки. Пример 17.20 будет последней проверкой графов перед тем, как двинуться дальше; набросайте узлы и дуги на бумаге, если вам труднее, чем Python, проследить пути.

Пример 17.20. PP2E\Dstruct\Classics\gtestobj2.py

```
from graph import Graph

S = Graph('s')
P = Graph('p')       # граф spam
A = Graph('a')       # создать объекты узлов
M = Graph('m')

S.arcs = [P, M]      # S ведет к P и M
P.arcs = [S, M, A]   # дуги: вложенные объекты
A.arcs = [M]
print S.search(M)    # найти все пути из S в M
```

Этот тест находит в графе три пути между узлами S и M. Если хотите посмотреть еще примеры кода Python, связанного с графами, выберите файлы из каталога CD MoreGraphs. Они примерно такие же, как перечисленные здесь, но дополнительно вводят взаимодействие с пользователем при нахождении каждого решения. Кроме того, мы лишь слегка коснулись здесь этой области; поищите в других книгах дополнительные темы (например, поиск в ширину и по первому наилучшему совпадению):

```
C:\...\PP2E\Dstruct\Classics>python gtestobj2.py
[[s, m], [s, p, m], [s, p, a, m]]
```

Реверсирование последовательностей

Реверсирование последовательностей является еще одной типичной операцией. В Python можно запрограммировать его рекурсивно или итеративно, а также в виде функций или методов классов. Пример 17.21 служит первой пробой двух простых функций реверсирования.

Пример 17.21. PP2E\Dstruct\Classics\rev1.py

```
def reverse(list):               # рекурсивно
    if list == []:
        return []
    else:
        return reverse(list[1:]) + list[:1]
def ireverse(list):              # итеративно
    res = []
    for x in list: res = [x] + res
    return res
```

Обе функции реверсирования корректно обрабатывают списки. Но если попробовать реверсировать последовательности, не являющиеся списками (строки, кортежи и т. п.), происходят неприятности: функция `ireverse` всегда возвращает в качестве результата список независимо от того, какая последовательность ей передана:

```
>>> ireverse("spam")
['m', 'a', 'p', 's']
```

Еще хуже, что рекурсивная версия `reverse` вообще работает только со списками – с другими последовательностями она входит в бесконечный цикл. Причина этого тонкая: когда `reverse` добирается до пустой строки (`''`), последняя оказывается не равной пустому списку (`[]`), поэтому выбирается предложение `else`. Но срез пустой последовательности снова возвращает пустую последовательность (индексы увеличиваются): снова повторяется предложение `else` с пустой строкой и без возбуждения исключительной ситуации. В итоге функция попадает в бесконечный цикл, вызывая себя снова и снова, пока Python не исчерпает всю память.

Версии примера 17.22 исправляют обе проблемы, используя общую технику работы с последовательностями:

- `reverse` использует оператор `not` для обнаружения конца последовательности и возвращает саму пустую последовательность, а не константу пустого списка. Поскольку пустая последовательность имеет тип исходного аргумента, операция `+` всегда строит последовательность правильного типа по мере развертывания рекурсии.
- `ireverse` использует то обстоятельство, что срез последовательности возвращает последовательность того же типа. Она сначала инициализирует результат срезом `[:0]` – новым пустым срезом того же типа, что и аргумент. Затем с помощью срезов извлекаются последовательности из одного узла, добавляемые в начало результата, а не к константе списка.

Пример 17.22. PP2E\Dstruct\Classics\rev2.py

```
def reverse(list):
    if not list:                                # пусто? (не всегда [])
        return list                             # тот же тип последовательности
    else:
        return reverse(list[1:]) + list[:1]     # добавить передний элемент в конец

def ireverse(list):
    res = list[:0]                              # пусто, того же типа
    for i in range(len(list)):
        res = list[i:i+1] + res                 # добавить каждый элемент вперед
    return res
```

Эти функции работают с любыми последовательностями и возвращают новую последовательность того же типа, что и переданная в них. Если передать строку, получим в качестве результата новую строку. На самом деле они реверсируют любую последовательность объектов, которая поддерживает срезы, конкатенацию и `len` – даже экземпляров классов Python и типов C. Иными словами, они могут реверсировать любой объект с протоколами интерфейса последовательности. Вот пример их работы со списками, строками и кортежами:

```
% python
>>> from rev2 import *
>>> reverse([1,2,3]), ireverse([1,2,3])
([3, 2, 1], [3, 2, 1])
>>> reverse("spam"), ireverse("spam")
('maps', 'maps')
```

```
>>> reverse((1.2, 2.3, 3.4)), ireverse((1.2, 2.3, 3.4))
((3.4, 2.3, 1.2), (3.4, 2.3, 1.2))
```

Перестановки последовательностей

Функции, определенные в примере 17.23, предоставляют несколько способов перемешивания последовательностей:

- `permute` строит список всех возможных перестановок последовательности.
- `subset` строит список всех возможных перестановок указанной длины.
- `combo` действует, как `subset`, но порядок не имеет значения: перестановки из одинаковых элементов удаляются.

Эти результаты полезны в ряде алгоритмов: при поиске, статистическом анализе и др. Например, один из способов оптимально упорядочить элементы состоит в том, чтобы поместить их в список, сгенерировать всевозможные перестановки и просто по очереди проверить их. Все три функции используют общие приемы срезов последовательностей из функций реверсирования предыдущего раздела, чтобы результирующий список содержал последовательности того же типа, что и переданная в качестве аргумента (например, при перестановке строки мы получаем список строк).

Пример 17.23. PP2E\Dstruct\Classics\permcomb.py

```
def permute(list):
    if not list:                                    # перемешивание любой последовательности
        return [list]                               # пустая последовательность
    else:
        res = []
        for i in range(len(list)):
            rest = list[:i] + list[i+1:]            # удалить текущий узел
            for x in permute(rest):                 # перестановка остальных
                res.append(list[i:i+1] + x)         # добавить узел в начало
        return res

def subset(list, size):
    if size == 0 or not list:                       # здесь важен порядок
        return [list[:0]]                           # пустая последовательность
    else:
        result = []
        for i in range(len(list)):
            pick = list[i:i+1]                      # срез последовательности
            rest = list[:i] + list[i+1:]            # сохранить часть [:i]
            for x in subset(rest, size-1):
                result.append(pick + x)
        return result

def combo(list, size):
    if size == 0 or not list:                       # порядок не важен
        return [list[:0]]                           # xyz == yzx
    else:
        result = []
        for i in range(0, (len(list) - size) + 1):  # iff осталось достаточно
            pick = list[i:i+1]
            rest = list[i+1:]                       # отбросить часть [:i]
            for x in combo(rest, size - 1):
                result.append(pick + x)
        return result
```

Как и функции реверсирования, все эти три функции действуют над любым объектом, поддерживающим операции `len`, среза и конкатенации. Например, можно применить `permute` к экземплярам некоторых классов стеков, определенных в начале этой главы; обратно будет получен список объектов экземпляров стека с перемешанными узлами.

Вот несколько примеров применения наших функций перестановки. Перестановка списка позволяет найти все способы, которыми могут быть упорядочены элементы. Например, для списка из четырех элементов возможны 24 перестановки ($4 \times 3 \times 2 \times 1$). После выбора одного из четырех на первую позицию остается только три, из которых можно выбрать на вторую позицию, и т. д. Порядок имеет значение: `[1,2,3]` не то же самое, что `[1,3,2]`, поэтому в результате появляются оба списка:

```
C:\...\PP2E\Dstruct\Classics>python
>>> from permcomb import *
>>> permute([1,2,3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
>>> permute('abc')
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> permute('help')
['help', 'hepl', 'hlep', 'hlpe', 'hpel', 'hple', 'ehlp', 'ehpl', 'elhp', 'elph',
 'ephl', 'eplh', 'lhep', 'lhpe', 'lehp', 'leph', 'lphe', 'lpeh', 'phel', 'phle',
 'pehl', 'pelh', 'plhe', 'pleh']
```

Результаты `combo` близки к перестановкам, но на результат накладывается ограничение фиксированной длины, а порядок не имеет значения: `abc` есть то же самое, что `acb`, поэтому только один элемент добавляется в результирующее множество:

```
>>> combo([1,2,3], 3)
[[1, 2, 3]]
>>> combo('abc', 3)
['abc']
>>> combo('abc', 2)
['ab', 'ac', 'bc']
>>> combo('abc', 4)
[]
>>> combo((1, 2, 3, 4), 3)
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
>>> for i in range(0, 6): print i, combo("help", i)
...
0 ['']
1 ['h', 'e', 'l', 'p']
2 ['he', 'hl', 'hp', 'el', 'ep', 'lp']
3 ['hel', 'hep', 'hlp', 'elp']
4 ['help']
5 []
```

Наконец, `subset` представляет просто перестановки фиксированной длины; порядок имеет значение, поэтому результат больше по объему, чем для `combo`. На самом деле вызов `subset` с длиной последовательности идентичен `permute`:

```
>>> subset([1,2,3], 3)
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
>>> subset('abc', 3)
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> for i in range(0, 6): print i, subset("help", i)
...
0 ['']
```

```
1 ['h', 'e', 'l', 'p']
2 ['he', 'hl', 'hp', 'eh', 'el', 'ep', 'lh', 'le', 'lp', 'ph', 'pe', 'pl']
3 ['hel', 'hep', 'hle', 'hlp', 'hpe', 'hpl', 'ehl', 'ehp', 'elh', 'elp', 'eph',
   'epl', 'lhe', 'lhp', 'leh', 'lep', 'lph', 'lpe', 'phe', 'phl', 'peh', 'pel',
   'plh', 'ple']
4 ['help', 'hepl', 'hlep', 'hlpe', 'hpel', 'hple', 'ehlp', 'ehpl', 'elhp',
   'elph', 'ephl', 'eplh', 'lhep', 'lhpe', 'lehp', 'leph', 'lphe', 'lpeh',
   'phel', 'phle', 'pehl', 'pelh', 'plhe', 'pleh']
5 ['help', 'hepl', 'hlep', 'hlpe', 'hpel', 'hple', 'ehlp', 'ehpl', 'elhp',
   'elph', 'ephl', 'eplh', 'lhep', 'lhpe', 'lehp', 'leph', 'lphe', 'lpeh',
   'phel', 'phle', 'pehl', 'pelh', 'plhe', 'pleh']
```

Сортировка последовательностей

Еще одним главным занятием многих систем является сортировка: упорядочение элементов совокупности в соответствии с некоторым ограничением. Пример сценария 17.24 определяет простую процедуру сортировки на Python, которая упорядочивает список объектов по полю. Поскольку индексирование в Python является общим, поле может быть индексом или ключом – эта функция может сортировать списки последовательностей или отображений.

Пример 17.24. PP2E\Dstruct\Classics\sort1.py

```
def sort(list, field):
    res = []                                    # всегда возвращает список
    for x in list:
        i = 0
        for y in res:
            if x[field] <= y[field]: break      # узел списка попадает сюда?
            i = i+1
        res[i:i] = [x]                          # вставить в слот результата
    return res

if __name__ == '__main__':
    table = [ {'name':'john', 'age':25}, {'name':'doe', 'age':32} ]
    print sort(table, 'name')
    print sort(table, 'age')
    table = [ ('john', 25), ('doe', 32) ]
    print sort(table, 0)
    print sort(table, 1)
```

Вот результат выполнения кода самотестирования этого модуля:

```
C:\...\PP2E\Dstruct\Classics>python sort1.py
[{'age': 32, 'name': 'doe'}, {'age': 25, 'name': 'john'}]
[{'age': 25, 'name': 'john'}, {'age': 32, 'name': 'doe'}]
[('doe', 32), ('john', 25)]
[('john', 25), ('doe', 32)]
```

Добавление функций сравнения

Поскольку функцию можно передавать в качестве параметра, как любой другой объект, легко предусмотреть дополнительное задание функции сравнения. В следующей версии (пример 17.25) второй аргумент принимает функцию, которая возвращает значение «истина», если ее первый аргумент должен быть помещен перед вторым. По умолчанию используется `lambda`, обеспечивающая порядок по возрастанию. Этот сортировщик тоже возвращает новую последовательность того же типа, что и переданная

ему, применяя такую же технику срезов, как в предыдущих разделах: если сортируется кортеж узлов, назад возвращается тоже кортеж.

Пример 17.25. PP2E\Dstruct\Classics\sort2.py

```
def sort(seq, func=(lambda x,y: x <= y)):      # по умолчанию: возрастание
    res = seq[:0]                              # возвратить тип seq
    for j in range(len(seq)):
        i = 0
        for y in res:
            if func(seq[j], y): break
            i = i+1
        res = res[:i] + seq[j:j+1] + res[i:]   # seq может быть неизменяемой
    return res

if __name__ == '__main__':
    table = ({'name':'doe'}, {'name':'john'})
    print sort(list(table),  (lambda x, y: x['name'] > y['name']))
    print sort(tuple(table), (lambda x, y: x['name'] <= y['name']))
    print sort('axbyzc')
```

На этот раз записи таблицы упорядочены в соответствии с переданной функцией сравнения полей:

```
C:\...\PP2E\Dstruct\Classics>python sort2.py
[{'name': 'john'}, {'name': 'doe'}]
({'name': 'doe'}, {'name': 'john'})
abcxyz
```

Эта версия обходится вообще без понятия поля и позволяет переданной функции при необходимости работать с индексами. В результате данная версия становится гораздо более общей; например, она также полезна для сортировки строк.

Структуры данных в сравнении со встроенными типами Python

Теперь, показав вам все эти сложные алгоритмы, я должен также сообщить, что не во всех случаях они дают оптимальный подход. Такие встроенные типы, как списки и словари, часто представляют данные более простым и эффективным способом. Например:

Двоичные деревья

Они могут оказаться полезными во многих приложениях, но словари Python уже предоставляют высоко оптимизированный, написанный на C инструмент поиска в таблицах. Доступ к словарю по ключу, скорее всего, окажется быстрее, чем поиск в структуре дерева, кодируемого на Python:

```
>>> x = {}
>>> for i in [3,1,9,2,7]: x[i] = None                # вставка
>>> for i in range(10): print (i, x.has_key(i)),     # поиск
(0, 0) (1, 1) (2, 1) (3, 1) (4, 0) (5, 0) (6, 0) (7, 1) (8, 0) (9, 1)
```

Поскольку словари являются встроенным средством языка, они всегда доступны и обычно оказываются быстрее, чем основанные на Python реализации структур данных.

Алгоритмы графов

Они служат многим целям, но в некоторых приложениях написанная на чистом Python реализация очень большого графа может оказаться менее эффективной, чем требуется. Программам для графов может потребоваться максимальная производительность. Использование словарей вместо экземпляров классов для представления графов может несколько повысить производительность, но можно добиться этого и с помощью присоединенных откомпилированных расширений.

Алгоритмы сортировки

Они тоже являются важной частью многих программ, но метод `sort` встроенных списков Python столь быстр, что в большинстве случаев будет трудновато побить его, программируя на Python. На самом деле обычно лучше преобразовывать последовательности в списки, чтобы иметь возможность использовать встроенную сортировку :

```
temp = list(sequence)
temp.sort()
...использовать элементы в temp...
```

Для специальной сортировки передайте свою функцию сравнения:

```
>>> L = [{'n':3}, {'n':20}, {'n':0}, {'n':9}]
>>> L.sort( lambda x, y: cmp(x['n'], y['n']) )
>>> L
[{'n': 0}, {'n': 3}, {'n': 9}, {'n': 20}]
```

Алгоритмы реверсирования

Обычно они излишни по той же причине – списки Python предоставляют быстрый метод `reverse`, поэтому, возможно, правильнее сначала преобразовать последовательность в список, чтобы воспользоваться его встроенным методом.

Поймите правильно: иногда действительно нужны объекты, которые добавляют новые функции к встроенным типам или выполняют что-то более индивидуальное. Например, классы множеств, с которыми мы познакомились, вводят инструменты, не поддерживаемые сегодня в Python непосредственно, а реализация стека с помощью деревьев кортежей оказалась действительно более быстрой, чем основанная на встроенных списках, в стандартных схемах применения. Перестановки тоже требуется добавлять самостоятельно.

Кроме того, инкапсуляция в классах позволяет изменять и расширять внутреннее устройство объекта, не трогая остальной части системы. Классы также значительно лучше поддерживают повторное использование кода, чем встроенные типы – типы на сегодняшний день не являются классами и не могут непосредственно специализироваться без логики классов-оболочек.

Однако поскольку в Python есть набор встроенных гибких и оптимизированных типов данных, реализации структур данных в Python часто не так важны, как в хуже оснащенных языках, таких как C или C++. Прежде чем начать писать новый тип данных, обязательно спросите себя, не будет ли встроенный тип или вызов лучше согласовываться с идеями Python.

PyTree: общее средство просмотра деревьев объектов

До сего момента эта глава ориентировалась на использование командной строки. Чтобы завершить ее, я хочу показать программу, в которой объединены технология GUI,

изучавшаяся ранее, и некоторые идеи структур данных, с которыми мы познакомились в этой главе.

Эта программа называется *PyTree* и служит общим средством просмотра древовидных структур данных, написанным на Python с применением библиотеки GUI Tkinter. PyTree рисует на экране узлы дерева в виде прямоугольников, соединенных стрелками. Она также умеет пересылать в дерево щелчки мыши по изображениям узлов, чтобы вызывать специфические для дерева действия. Так как PyTree позволяет визуализировать структуру дерева, генерируемого набором параметров, это интересный способ исследования основанных на деревьях алгоритмов.

PyTree поддерживает произвольные типы деревьев, «оборачивая» реальные деревья в объекты интерфейса. Объекты интерфейса реализуют стандартный протокол путем обмена данными с базовым объектом дерева. Для целей данной главы PyTree оснащена средствами отображения *двоичных деревьев поиска*; она также может выводить *деревья синтаксического разбора выражений*, что понадобится в следующей главе. Просмотр других деревьев осуществляется кодированием классов-оболочек для взаимодействия с новыми типами деревьев.

Интерфейсы GUI, используемые в PyTree, в полной мере освещались ранее в этой книге, поэтому я не стану здесь подробно останавливаться на этом коде (см. детали в части II «Программирование GUI»). Постарайтесь запустить эту программу на своем компьютере, чтобы получить лучшее представление об ее работе. Так как она написана на Python с Tkinter, то должна быть переносимой под Windows, Unix и Macintosh.

Работа с PyTree

Прежде чем браться за код, посмотрим, как выглядит PyTree. Запустить PyTree можно из панели запуска PyDemos (см. верхний уровень дерева исходных кодов дистрибутива примеров на CD, прилагаемом к книге) или непосредственно как файл *treeview.py*, приведенный в примере 17.27. Рис. 17.2 показывает PyTree в работе, выводящей двоичное дерево, созданное по нажатию кнопки «test1». Деревья изображаются как метки, помещаемые на холст и соединяемые линиями со стрелками. Линии отражают отношение «родитель-потомок» в фактическом дереве; PyTree пытается расположить дерево так, чтобы получить более или менее единообразное отображение типа показанного.

Окно PyTree состоит из холста с вертикальной и горизонтальной полосами прокрутки и ряда управляющих элементов в нижней части – переключателей для выбора типа дерева, которое нужно отобразить, набора кнопок, запускающих тесты отображения готовых деревьев, и текстового поля ввода для задания и генерации нового дерева. Набор кнопок тестов изменяется, если выбрать переключатель Parser (тестовых кнопок становится на одну меньше); PyTree использует методы графических элементов `pack_forget` и `pack`, чтобы скрывать или показывать специфические для деревьев кнопки на лету.

При выборе одной из кнопок готовых тестов в поле ввода показывается строка, которую следовало бы набрать для генерации отображенного дерева. Для бинарных деревьев введите список значений, разделенных пробелами, и нажмите кнопку «input» или клавишу Enter, чтобы сгенерировать новое дерево; новое дерево является результатом вставки введенных значений слева направо. Для деревьев синтаксического разбора нужно ввести в поле ввода строку выражения (подробнее об этом позднее). Рис. 17.3 показывает результат ввода и передачи программе ряда значений в поле ввода; результирующее двоичное дерево появляется на холсте.

Обратите внимание на всплывающее окно на этом снимке экрана; щелчок левой кнопкой мыши по узлу отображаемого дерева запускает действие, определенное в классе-

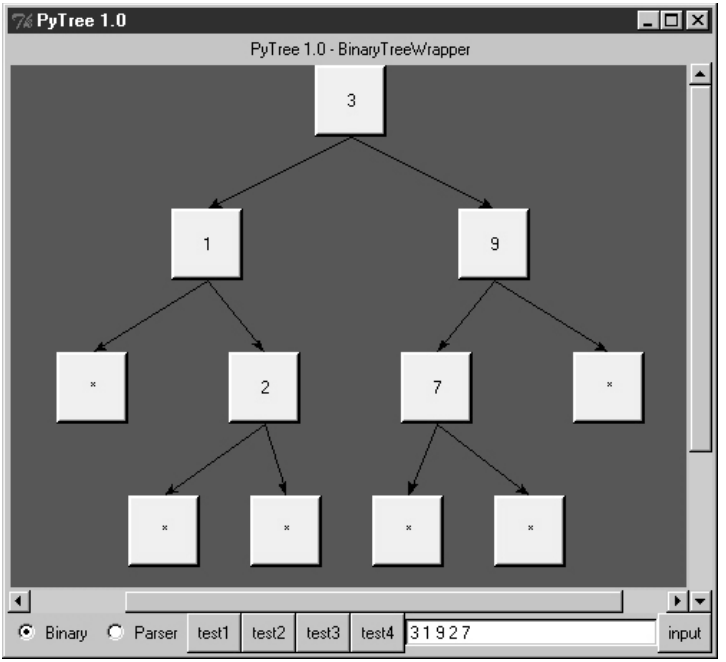


Рис. 17.2. Просмотр бинарного дерева поиска в PyTree (test1)

оболочке дерева, и показывает его результат во всплывающем окне. Для двоичных деревьев нет выполняемого действия, поэтому во всплывающем окне показывается сообщение по умолчанию, но в деревьях синтаксического анализа по щелчку мыши вычисляется поддерево, корнем которого служит щелкнутый узел (опять-таки, о деревьях синтаксического разбора потом).

Для развлечения увеличьте окно на весь экран и нажмите кнопку «test4» – в результате 100 чисел от 0 до 99 будут вставлены в новое двоичное дерево случайным образом, и показан результат. На рис. 17.4 схвачена одна часть этого дерева; оно слишком велико, чтобы уместиться на одном экране (или одной странице книги), но можно перемещаться по дереву с помощью полос прокрутки холста.

PyTree использует специальный алгоритм, чтобы в этом дереве все родители соединялись с потомками непересекающимися линиями. Он делает некоторый предварительный анализ, пытаясь на каждом уровне разместить потомков возможно ближе к родителям. На этом этапе анализа также вычисляется общий размер нового дерева – PyTree использует его для того, чтобы заново устанавливать размер прокручиваемой области холста для каждого отображаемого дерева.

Исходный код Pytree

Перейдем к коду. Аналогично PyForm из предыдущей главы, код PyTree размещен в двух модулях; здесь один модуль решает задачу изображения деревьев в GUI, а другой реализует оболочки для взаимодействия с различными типами деревьев и расширяет GUI дополнительными графическими элементами.

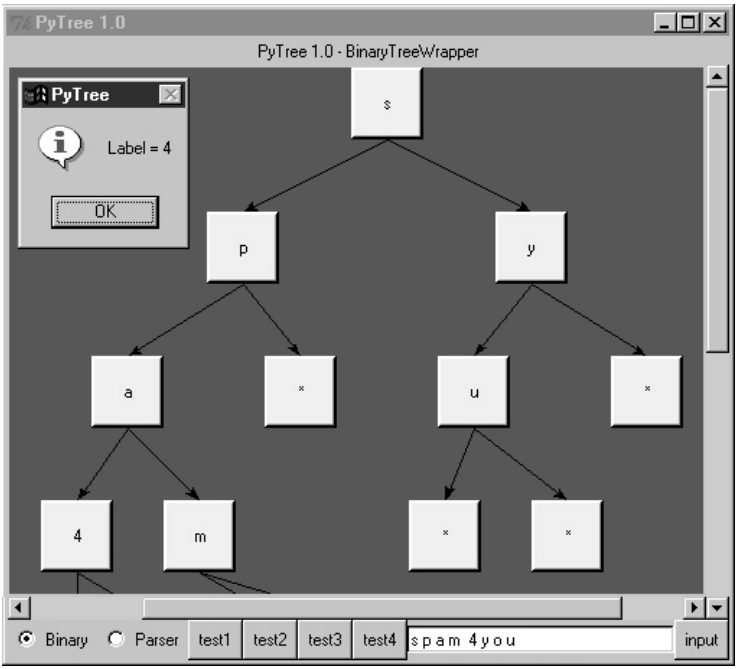


Рис. 17.3. Бинарное дерево, введенное вручную, и всплывающее при щелчке окно

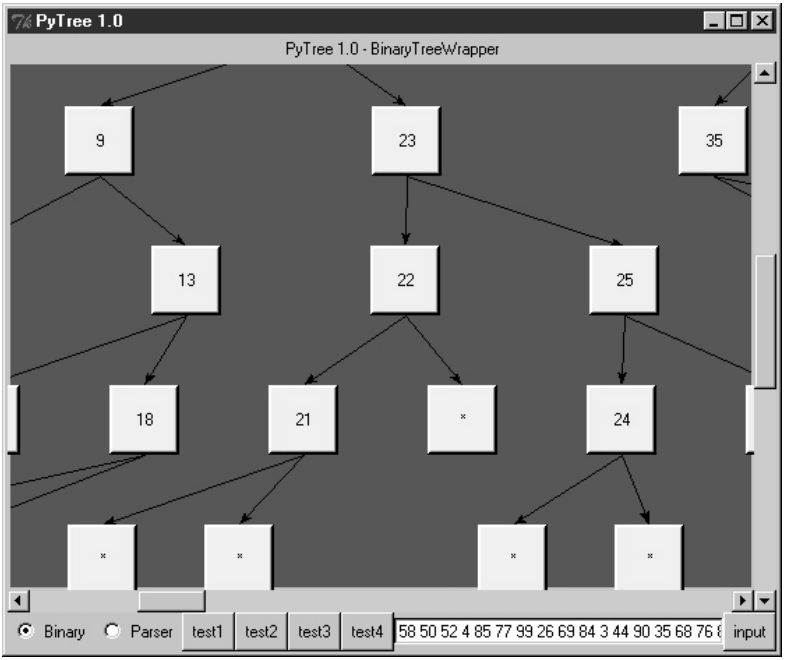


Рис. 17.4. Просмотр большого бинарного дерева поиска в PyTree (test4)

Независимая от деревьев реализация GUI

Модуль в примере 17.26 выполняет действия, связанные с вычерчиванием деревьев на холсте. Он написан так, чтобы не зависеть от конкретной структуры дерева – его класс `TreeViewer` передает полномочия классу `TreeWrapper`, когда для рисования требуется специфическая для дерева информация (например, текст меток узлов, связи с дочерними узлами). `TreeWrapper`, в свою очередь, предполагает создание на его основе подклассов для конкретного типа дерева; в действительности он возбуждает ошибки утверждения при попытке использовать его без создания подклассов. Конструктивно `TreeWrapper` встраивается в `TreeViewer`; почти столь же просто кодировать классы `TreeViewer` для каждого типа дерева, но при этом GUI ограничивается одним конкретным типом дерева (см. основанную на подклассах альтернативу в *treeview_subclasses.py* на прилагаемом CD).

Деревья рисуются в два этапа – планирующий обход, во время которого строится структура данных расположения, связывающая родителей и потомков, и этап вычерчивания, на котором созданный план используется для рисования и связывания меток узлов на холсте. Такой двухэтапный подход частично упрощает логику, необходимую для единообразного расположения деревьев. Детали можно выяснить, изучая листинг.

Пример 17.26. PP2E\Dstruct\TreeView\treeview_wrappers.py

```
##############################################################################################
# PyTree: нарисовать структуры данных произвольного дерева на холсте с прокруткой; в этой
# версии используются классы-оболочки дерева, встраиваемые в gui вьюера, чтобы поддерживать
# произвольные деревья (т. е. композиция, а не подклассы вьюера); кроме того, добавлены обратные
# вызовы при щелчках по меткам узлов дерева – выполнение специфических для дерева действий;
# альтернативная структура, основанная на подклассах, есть в treeview_subclasses.py; создание
# подклассов ограничивает средство просмотра одним типом дерева, оболочки - нет; альтернативный
# способ рисования дерева объекта есть в treeview_left.py; контрольные примеры для двоичного
# дерева и дерева синтаксического анализа можно посмотреть и выполнить в treeview.py;
##############################################################################################

from Tkinter import *
from tkMessageBox import showinfo

Width, Height = 350, 350     # начальный размер холста (переустанавливается для каждого дерева)
Rowsz = 100                                 # пикселов в строке дерева
Colsz = 100                                 # пикселов в колонке дерева

####################################
# интерфейс к узлам объекта дерева
####################################

class TreeWrapper:                          # подкласс для типа дерева
    def children(self, treenode):
        assert 0, 'children method must be specialized for tree type'
    def label(self, treenode):
        assert 0, 'label method must be specialized for tree type'
    def value(self, treenode):
        return ''
    def onClick(self, treenode):            # обратный вызов для щелчка по метке узла
        return ''
    def onInputLine(self, line, viewer):    # обратный вызов отправки входной строки
        pass

##################################
# tree view gui, tree independent
##################################
```

```
class TreeViewer(Frame):
    def __init__(self, wrapper, parent=None, tree=None, bg='brown', fg='beige'):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)
        self.makeWidgets(bg)                # создание gui: холст с прокруткой
        self.master.title('PyTree 1.0')     # предполагается автономное выполнение
        self.wrapper = wrapper              # встроить объект TreeWrapper
        self.fg = fg                        # setTreeType изменяет оболочку
        if tree:
           self.drawTree(tree)

    def makeWidgets(self, bg):
        self.title = Label(self, text='PyTree 1.0')
        self.canvas = Canvas(self, bg=bg, borderwidth=0)
        vbar = Scrollbar(self)
        hbar = Scrollbar(self, orient='horizontal')

        self.title.pack(side=TOP, fill=X)
        vbar.pack(side=RIGHT,  fill=Y)                  # упаковка холста после полос
        hbar.pack(side=BOTTOM, fill=X)
        self.canvas.pack(side=TOP, fill=BOTH, expand=YES)

        vbar.config(command=self.canvas.yview)          # вызов при движении прокрутки
        hbar.config(command=self.canvas.xview)
        self.canvas.config(yscrollcommand=vbar.set)     # вызов при движении холста
        self.canvas.config(xscrollcommand=hbar.set)
        self.canvas.config(height=Height, width=Width)  # размер видимой области

    def clearTree(self):
        mylabel = 'PyTree 1.0 - ' + self.wrapper.__class__.__name__
        self.title.config(text=mylabel)
        self.unbind_all('<Button-1>')
        self.canvas.delete('all')                       # очистка событий, рисование

    def drawTree(self, tree):
        self.clearTree()
        wrapper = self.wrapper
        levels, maxrow = self.planLevels(tree, wrapper)
        self.canvas.config(scrollregion=(                       # область прокрутки
            0, 0, (Colsz * maxrow), (Rowsz * len(levels)) ))  # левый верхний, правый нижний
        self.drawLevels(levels, maxrow, wrapper)

    def planLevels(self, root, wrap):
        levels = []
        maxrow = 0                                              # обход дерева для
        currlevel = [(root, None)]                              # размещения строк, колонок
        while currlevel:
            levels.append(currlevel)
            size = len(currlevel)
            if size > maxrow: maxrow = size
            nextlevel = []
            for (node, parent) in currlevel:
                if node != None:
                    children = wrap.children(node)              # список узлов
                    if not children:
                        nextlevel.append((None, None))          # оставить дырку
                    else:
                        for child in children:
                            nextlevel.append((child, node))     # связь с родителем
            currlevel = nextlevel
```

```
        return levels, maxrow

    def drawLevels(self, levels, maxrow, wrap):
        rowpos = 0                                          # чертить дерево по плану
        for level in levels:                                # установить обработчики щелчков
            colinc = (maxrow * Colsz) / (len(level) + 1)    # levels – узлы дерева
            colpos = 0
            for (node, parent) in level:
                colpos = colpos + colinc
                if node != None:
                    text = wrap.label(node)
                    more = wrap.value(node)
                    if more: text = text + '=' + more
                    win = Label(self.canvas, text=text, bg=self.fg, bd=3, relief=RAISED)
                    win.pack()
                    win.bind('<Button-1>',
                             lambda e, n=node, handler=self.onClick: handler(e, n))
                    self.canvas.create_window(colpos, rowpos, anchor=NW,
                             window=win, width=Colsz*.5, height=Rowsz*.5)
                    if parent != None:
                        self.canvas.create_line(
                            parent.__colpos + Colsz*.25,    # от x-y, до x-y
                            parent.__rowpos + Rowsz*.5,
                            colpos + Colsz*.25, rowpos, arrow='last', width=1)
                    node.__rowpos = rowpos
                    node.__colpos = colpos       # пометить узел, закрытые атрибуты
            rowpos = rowpos + Rowsz

    def onClick(self, event, node):
        label = event.widget
        wrap  = self.wrapper
        text  = 'Label = ' + wrap.label(node)    # при щелчке по метке
        value = wrap.value(node)
        if value:
            text = text + '\nValue = ' + value   # добавить текст дерева
        result = wrap.onClick(node)              # выполнить действие дерева
        if result:
            text = text + '\n' + result          # добавить результат действия
        showinfo('PyTree', text)                 # показ стандартного диалога

    def onInputLine(self, line):                 # передать текст оболочке дерева
        self.wrapper.onInputLine(line, self)     # пример: анализ и перерисовка дерева
    def setTreeType(self, newTreeWrapper):       # изменить тип дерева
        if self.wrapper != newTreeWrapper:       # действует при следующем рисовании
            self.wrapper = newTreeWrapper
            self.clearTree()                     # или старый узел, новая оболочка
```

Оболочки деревьев и test widgets

Вторую половину PyTree составляет модуль, определяющий подклассы `TreeWrapper`, взаимодействующие с деревьями – двоичными и синтаксического анализа, реализует кнопки готовых контрольных примеров и добавляет управляющие элементы в нижнюю часть окна PyTree.[1] Эти управляющие элементы специально были выделены в

[1] Если хотите поупражняться в программировании, попробуйте добавить другой класс-оболочку и переключатель для просмотра `KeyedBinaryTree`, которое мы написали ранее в этой главе. Вероятно, нужно показать ключ в GUI и выводить окно с ассоциированным значением при щелчках.

этот отдельный модуль (пример 17.27), потому что холст PyTree может оказаться полезным в качестве средства просмотра компонентов в других приложениях GUI.

Пример 17.27. PP2E\Dstruct\TreeView\treeview.py

```
# Сценарий запуска PyTree
# оболочки для просмотра деревьев типов, имеющихся в книге, плюс контрольные примеры/gui

import string
from Tkinter import *
from treeview_wrappers import TreeWrapper, TreeViewer
from PP2E.Dstruct.Classics import btree
from PP2E.Lang.Parser import parser2

#########################################################################
# оболочка двоичных деревьев
#########################################################################

class BinaryTreeWrapper(TreeWrapper):     # встроить двоичное дерево в viewer
    def children(self, node):             # добавляет протоколы просмотра
        try:                              # для взаимодействия с деревом
            return [node.left, node.right]
        except:
            return None
    def label(self, node):
        try:
            return str(node.data)
        except:
            return str(node)
    def onInputLine(self, line, viewer):   # при вводе внизу
        items = string.split(line)         # создать дерево из текстовых данных
        t = btree.BinaryTree()             # нарисовать полученное btree
        for x in items: t.insert(x)        # здесь нет обработчика onClick
        viewer.drawTree(t.tree)

#########################################################################
# расширение двоичного дерева
#########################################################################

class BinaryTree(btree.BinaryTree):
    def __init__(self, viewer):                # встроить viewer в дерево
        btree.BinaryTree.__init__(self)        # но у viewer есть оболочка
        self.viewer  = viewer
    def view(self):
        self.viewer.drawTree(self.tree)

#########################################################################
# оболочка дерева синтаксического разбора
#########################################################################

class ParseTreeWrapper(TreeWrapper):
    def __init__(self):                        # встроить дерево разбора в viewer
        self.dict = {}                         # добавить протоколы вьюера
    def children(self, node):
        try:
            return [node.left, node.right]
        except:
            try:
                return [node.var, node.val]
            except:
                return None
    def label(self, node):
```

```
        for attr in ['label', 'num', 'name']:
            if hasattr(node, attr):
                return str(getattr(node, attr))
        return 'set'
    def onClick(self, node):                  # при щелчке по метке в дереве
        try:                                  # специфическое для дерева действие
            result = node.apply(self.dict)    # вычислить поддерево
            return 'Value = ' + str(result)   # показать результат во всплывающем окне
        except:
            return 'Value = <error>'
    def onInputLine(self, line, viewer):      # при вводе строки
        p = parser2.Parser()                  # разобрать текст выражения
        p.lex.newtext(line)                   # нарисовать полученное дерево
        t = p.analyse()
        if t: viewer.drawTree(t)

##############################################################################################
# готовые контрольные примеры (или введите новые списки узлов/выражения в поле ввода)
##############################################################################################

def shownodes(sequence):
    sequence = map(str, sequence)           # преобразовать узлы в строки
    entry.delete(0, END)                    # показать узлы в текстовом поле
    entry.insert(0, string.join(sequence, ' '))

def test1_binary():                         # оболочка для двоичного дерева
    nodes = [3, 1, 9, 2, 7]                 # создать двоичное дерево
    tree  = BinaryTree(viewer)              # встроить viewer в дерево
    for i in nodes: tree.insert(i)
    shownodes(nodes)                        # показать узлы в поле ввода
    tree.view()                             # нарисовать дерево через встроенный viewer

def test2_binary():
    nodes = 'badce'
    tree  = btree.BinaryTree()              # встроить оболочку в viewer
    for c in nodes: tree.insert(c)          # создать двоичное дерево
    shownodes(nodes)
    viewer.drawTree(tree.tree)              # попросить viewer нарисовать его

def test3_binary():
    nodes = 'abcde'
    tree  = BinaryTree(viewer)
    for c in nodes: tree.insert(c)
    shownodes(nodes)
    tree.view()

def test4_binary():
    tree = BinaryTree(viewer)
    import random                           # сделать большое двоичное дерево
    nodes = range(100)                      # вставить случайные 100 узлов
    order = []                              # и нарисовать в viewer
    while nodes:
        item = random.choice(nodes)
        nodes.remove(item)
        tree.insert(item)
        order.append(item)
    shownodes(order)
    tree.view()

def test_parser(expr):
    parser = parser2.Parser()              # оболочка для синтаксического дерева
```

```
    parser.lex.newtext(expr)              # поддеревья вычисляются при щелчке
    tree   = parser.analyse()             # строка ввода разбирает новое выражение
    entry.delete(0, END)                  # переменные установлены в словаре оболочки
    entry.insert(0, expr)                 # см. анализатор в главе по языкам/тексту
    if tree: viewer.drawTree(tree)

def test1_parser(): test_parser("1 + 3 * (2 * 3 + 4)")
def test2_parser(): test_parser("set temp 1 + 3 * 2 * 3 + 4")
def test3_parser(): test_parser("set result temp + ((1 + 3) * 2) * (3 + 4)")

#################################################################
# построить вьюер (средство просмотра) с дополнительными графическими
# элементами для тестирования типов деревьев
#################################################################

if __name__ == '__main__':
    root = Tk()                           # создать единый gui просмотра
    bwrapper = BinaryTreeWrapper()        # дополнительно: строка ввода, кнопки тестов
    pwrapper = ParseTreeWrapper()         # создать объекты оболочек
    viewer   = TreeViewer(bwrapper, root) # начать в двоичном режиме

    def onRadio():
        if var.get() == 'btree':
            viewer.setTreeType(bwrapper)          # изменить оболочку
            for btn in p_btns: btn.pack_forget()  # убрать кнопки тестов анализатора
            for btn in b_btns: btn.pack(side=LEFT)# показать двоичные кнопки
        elif var.get() == 'ptree':
            viewer.setTreeType(pwrapper)
            for btn in b_btns: btn.pack_forget()
            for btn in p_btns: btn.pack(side=LEFT)

    var = StringVar()
    var.set('btree')
    Radiobutton(root, text='Binary', command=onRadio, variable=var, value='btree').pack(side=LEFT)
    Radiobutton(root, text='Parser', command=onRadio, variable=var, value='ptree').pack(side=LEFT)
    b_btns = []
    b_btns.append(Button(root, text='test1', command=test1_binary))
    b_btns.append(Button(root, text='test2', command=test2_binary))
    b_btns.append(Button(root, text='test3', command=test3_binary))
    b_btns.append(Button(root, text='test4', command=test4_binary))
    p_btns = []
    p_btns.append(Button(root, text='test1', command=test1_parser))
    p_btns.append(Button(root, text='test2', command=test2_parser))
    p_btns.append(Button(root, text='test3', command=test3_parser))
    onRadio()

    def onInputLine():
        line = entry.get()              # для текущего типа оболочки дерева
        viewer.onInputLine(line)        # ввод списка узлов или выражения

    Button(root, text='input', command=onInputLine).pack(side=RIGHT)
    entry = Entry(root)
    entry.pack(side=RIGHT, expand=YES, fill=X)
    entry.bind('<Return>', lambda event: onInputLine())   # кнопка или клавиша enter
    root.mainloop()                                       # запуск gui
```

Pytree тоже производит анализ деревьев

Наконец, я хочу показать, что происходит при щелчке по переключателю Parser в окне PyTree. GUI переключается на просмотр дерева синтаксического анализа выражения путем использования другого класса оболочки дерева: меняется метка вверху,

меняются кнопки тестов, а вводить теперь надо арифметическое выражение, которое должно быть проанализировано и нарисовано. Рис. 17.5 показывает дерево, сгенерированное для строки выражения, выведенной в поле ввода.

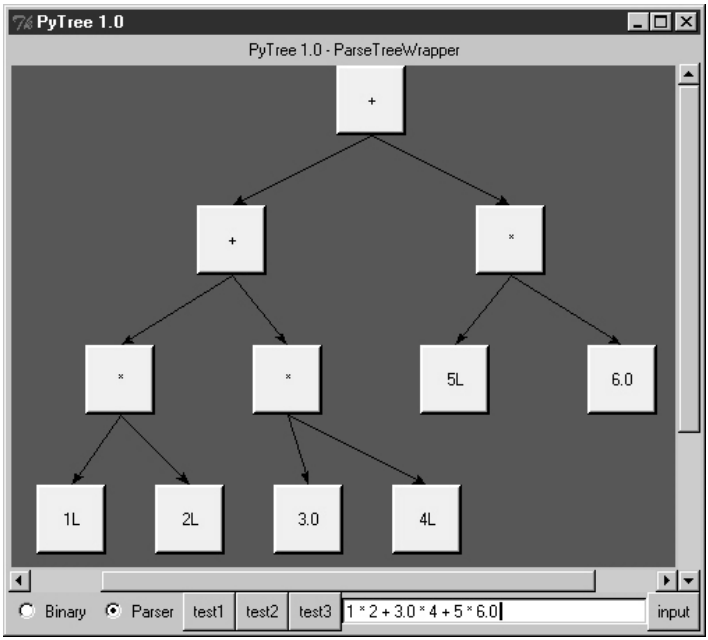


Рис. 17.5. Просмотр дерева синтаксического анализа выражения в PyTree

PyTree задумано как общее средство – оно выводит двоичные деревья и деревья синтаксического анализа, но легко ввести расширения для других типов деревьев с помощью новых оболочек классов. В GUI можно в любой момент переключиться между двоичным деревом и деревом синтаксического анализа щелчком по переключателю. Данные, вводимые в поле ввода, оцениваются в соответствии с текущим типом дерева. Когда просмотр ведется в режиме дерева синтаксического анализа, щелчок по узлу дерева вызывает вычисление части выражения, представленной деревом, корень которого находится в этом узле. Рис. 17.6 показывает всплывающее окно, выводимое при щелчке по корневому узлу показанного дерева.

При просмотре деревьев синтаксического анализа PyTree становится своего рода графическим калькулятором – можно генерировать произвольные деревья выражений и вычислять любую их часть, щелкая по показанным узлам. Но я не хочу рассказывать о деревьях этого типа, пока мы не перейдем к следующей главе.

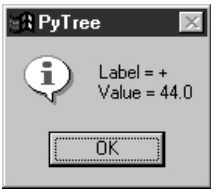


Рис. 17.6. Всплывающее окно PyTree после щелчка по узлу дерева синтаксического анализа

18

Текст и язык

«See Jack Hack. Hack, Jack, Hack»

Обработка текстовой информации в той или иной форме является одной из наиболее частых задач, которые приходится выполнять приложениям. Это может предполагать все – от просмотра текстового файла по колонкам до анализа утверждений языка, определяемого формальной грамматикой. Такую обработку обычно называют *синтаксическим анализом* (*parsing*) – разбором структуры текстовой строки. В этой главе мы изучим способы обработки языковой и текстовой информации, а попутно во врезках будут резюмированы некоторые концепции разработки на Python.

Часть материала является сложной, но примеры невелики. Например, синтаксический анализ методом рекурсивного спуска (recursive descent parsing) иллюстрируется простым примером, показывающим, как он может быть реализован в Python. Будет также показано, что часто нет необходимости писать специальные анализаторы для всех задач обработки языков в Python. Обычно вместо этого можно экспортировать API и использовать их в программах Python, при этом иногда достаточно одного вызова встроенной функции. Завершается глава представлением *PyCalc* – GUI-калькулятора, написанного на Python и являющегося последним крупным примером программы Python в этой книге. Как мы увидим, написание калькуляторов состоит в основном в жонглировании стеками во время лексического анализа текста.

Стратегии синтаксического анализа в Python

В общем плане есть ряд подходов к обработке текста в Python:

- Встроенные строковые объекты
- Средства модуля string (и методов строк)
- Регулярные выражения
- Интеграция с генератором программ синтаксического анализа (парсеров)
- Программы синтаксического анализа (парсеры) собственного написания
- Выполнение кода Python через встроенные функции `eval` и `exec`

Для простых задач в действительности часто достаточно встроенного объекта Python string. Со строками Python можно осуществлять обращение по индексу, конкатенацию, срезы и обрабатывать их с помощью встроенных функций и модуля `string`. Однако в этой главе делается упор на средствах более высокого уровня и технике анализа текстовой информации. Рассмотрим кратко каждый из этих подходов с характерными примерами.

Средства модуля string

Модуль Python `string` содержит ряд средств обработки текста, превосходящих возможности операторов строковых выражений. Например:

- `string.find` выполняет поиск подстрок.
- `string.atoi` преобразует строки в целые.
- `string.strip` удаляет ведущие и замыкающие пробельные символы.
- `string.upper` преобразует в верхний регистр.
- `string.replace` выполняет замену подстрок.

Руководство по библиотеке Python содержит полный список имеющихся средств. Кроме того, начиная с Python 2.0 средства работы со строками полностью поддерживают кодировку Unicode, а большинство функций модуля `string` теперь также доступны в качестве *методов* объекта строки. Например, в Python 2.0 следующие два выражения эквивалентны:

```
string.find(str, substr)      # обычный
str.find(substr)              # новый в 2.0
```

за исключением того, что во втором формате не требуется сначала импортировать модуль `string`. Как обычно, следует обратиться к библиотечным руководствам и приложению А «Последние изменения в Python», чтобы получить последние новости о средствах обработки строк.

Однако с учетом основной направленности этой главы особенно полезными для разбора текста оказываются встроенные средства Python для разбиения и соединения строк по элементам:

`string.split`

Разбивает строку на подстроки по пробельным символам (табуляция, пробел, перевод строки) или явно переданной строке-разделителю.

`string.join`

Сцепляет список или кортеж подстрок, добавляя между ними пробел или явно переданную строку-разделитель.

Как мы видели ранее в книге, `split` создает из строки список подстрок, а `join` снова соединяет их вместе:[1]

```
>>> import string
>>> string.split('A B C D')
['A', 'B', 'C', 'D']
>>> string.split('A+B+C+D', '+')
['A', 'B', 'C', 'D']
>>> string.join(['a', 'b', 'c'], '--')
'a--b--c'
```

Несмотря на простоту, они могут справляться с удивительно сложными задачами разбора текста. Кроме того, модуль `string` очень быстр, потому что перенесен на C. Например, чтобы быстро заменить все символы табуляции в файле четырьмя точками, направьте файл в стандартный ввод такого сценария:

---

[1] В прежних версиях Python были аналогичные средства с названиями `splitfields` и `joinfields`; более современные (и менее многословные) `split` и `join` сегодня являются более предпочтительным способом написания.

```
from sys import *
from string import *
stdout.write( join( split(stdin.read(), '\t'), '.'*4) )
```

Здесь вызов split делит входной поток по символам табуляции, а join соединяет его снова, вставляя четыре точки туда, где была табуляция. Объединение двух вызовов эквивалентно такому вызову глобальной замены в модуле string:

```
stdout.write( replace(stdin.read(), '\t', '.'*4) )
```

Суммирование по колонкам в файле

Рассмотрим пару практических применений расщепления и объединения строк. Во многих областях довольно часто возникает задача просмотра файлов по колонкам. Допустим, например, что имеется файл, содержащий колонки чисел, выведенных другой системой, и требуется сложить числа в каждой колонке. В Python это можно сделать с помощью разбиения строк (см. пример 18.1). Как дополнительный бонус, это решение нетрудно сделать повторно используемым в Python инструментом.

Пример 18.1. PP2E\Lang\summer.py

```
#!/usr/local/bin/python
import string

def summer(numCols, fileName):
    sums = [0] * numCols                            # создать список, заполненный нулями
    for line in open(fileName, 'r').readlines():    # просмотр строк файла
        cols = string.split(line)                   # разбить колонки
        for i in range(numCols):                    # по пробелам/табуляции
            sums[i] = sums[i] + eval(cols[i])       # сложить числа с sums
    return sums

if __name_== '__main__':
    import sys
    print summer(eval(sys.argv[1]), sys.argv[2])    # '% summer.py cols file'
```

Как обычно, можно *импортировать* этот файл и вызывать его функцию, а можно *запускать* его как инструмент оболочки из командной строки. Функция summer вызывает split, чтобы создать список строк, представляющих колонки строки, и eval для преобразования строк в колонках в числа. Вот входной файл, в котором для разделения колонок используются пробелы и табуляции:

```
C:\...\PP2E\Lang>type table1.txt
1       5       10    2    1.0
2       10      20    4    2.0
3       15      30    8     3
4       20      40   16    4.0
C:\...\PP2E\Lang>python summer.py 5 table1.txt
[10, 50, 100, 30, 10.0]
```

Обратите внимание, что поскольку сценарий суммирования использует eval для преобразования текста файла в числа, в действительности можно хранить в файле произвольные выражения Python. Вот пример выполнения сценария с файлом, содержащим фрагменты кода Python:

```
C:\...\PP2E\Lang>type table2.txt
2     1+1          1<<1          eval("2")
16    2*2*2*2      pow(2,4)      16.0
3     len('abc')   [1,2,3][2]    {'spam':3}['spam']
```

```
C:\...\PP2E\Lang>python summer.py 4 table2.txt
[21, 21, 21, 21.0]
```

Мы вернемся к eval далее в этой главе при рассмотрении вычисления выражений.[1]

Синтаксический разбор строк правил и обратное преобразование

Пример 18.2 демонстрирует один из способов применения разбиения и соединения строк для синтаксического разбора предложений простого языка. Он взят из оболочки основанной на правилах экспертной системы (holmes), которая написана на Python и включена в прилагаемый к книге CD (см. каталог примеров верхнего уровня *Ai*). Строки правил в holmes имеют вид:

```
"rule <id> if <test1>, <test2>... then <conclusion1>, <conclusion2>..."
```

Проверки и выводы являются конъюнкциями термов («,» означает «и»). Каждый терм является списком слов или переменных, разделенных пробелами; переменные начинаются с ?. Для использования правила оно переводится во внутренний формат – словарь с вложенными списками. Чтобы показать правило, оно переводится обратно в строковый формат. Например, для вызова

```
rules.internal_rule('rule x if a ?x, b then c, d ?x')
```

преобразование в функции internal_rule происходит так:

```
string = 'rule x if a ?x, b then c, d ?x'
i = ['rule x', 'a ?x, b then c, d ?x']
t = ['a ?x, b', 'c, d ?x']
r = ['', 'x']
result = {'rule':'x', 'if':[['a','?x'], ['b']], 'then':[['c'], ['d','?x']]}
```

Сначала происходит разбиение по if, затем по then и наконец по rule. Результат состоит из трех подстрок, разделенных по ключевым словам. Подстроки проверок и выводов разбиваются по «,» и, наконец, пробелам. join используется в обратном преобразовании (unparse) в исходную строку для отображения. Пример 18.2 является конкретной реализацией этой схемы.

Пример 18.2. PP2E\Lang\rules.py

```
from string import split, join, strip

def internal_rule(string):
    i = split(string, ' if ')
    t = split(i[1],   ' then ')
    r = split(i[0],   'rule ')
    return {'rule':strip(r[1]), 'if':internal(t[0]), 'then':internal(t[1])}

def external_rule(rule):
    return ('rule '    + rule['rule']           +
            ' if '     + external(rule['if'])   +
            ' then '   + external(rule['then']) + '.')

def internal(conjunct):
    res = []                                    # 'a b, c d'
    for clause in split(conjunct, ','):         # -> ['a b', ' c d']
```

[1] Смотрите также в примерах с сеткой в главе 8 «Обзор Tkinter, часть 2» использование механизма eval в таблицах. Приводимый здесь сценарий суммирования представляет значительно более простую версию логики суммирования по колонкам из этой главы.

```
            res.append(split(clause))              # -> [['a','b'], ['c','d']]
        return res

    def external(conjunct):
        strs = []
        for clause in conjunct:                    # [['a','b'], ['c','d']]
            strs.append(join(clause))              # -> ['a b', 'c d']
        return join(strs, ', ')                    # -> 'a b, c d'
```

Как обычно, можно проверить компоненты этого модуля интерактивно:

```
>>> import rules
>>> rules.internal('a ?x, b')
[['a', '?x'], ['b']]

>>> rules.internal_rule('rule x if a ?x, b then c, d ?x')
{'if': [['a', '?x'], ['b']], 'rule': 'x', 'then': [['c'], ['d', '?x']]}

>>> r = rules.internal_rule('rule x if a ?x, b then c, d ?x')
>>> rules.external_rule(r)
'rule x if a ?x, b then c, d ?x.'
```

Таким синтаксическим разбором путем разбиения строк по лексемам большего не достичь: нет прямой поддержки рекурсивной вложенности компонентов, а синтаксические ошибки обрабатываются не очень элегантно. Но для таких задач простых языков разбиения строк может оказаться достаточно, по крайней мере, для прототипирования систем. Потом всегда можно добавить более сильный анализатор правил или заново реализовать правила как встроенный код или классы Python.

Урок 1: делай прототип и переноси

Практика показывает, что по возможности нужно всюду использовать функции модуля `string` вместо таких вещей, как регулярные выражения. Обычно они действуют значительно быстрее, потому что их реализация переведена на C. При импорте `string` происходит внутренняя замена большей части его содержания функциями, импортированными из модуля расширения C `strop`; методы `strop` оказываются в 100–1000 раз быстрее, чем их эквиваленты на Python.[1]

Модуль `string` первоначально был написан на Python, но потребность в эффективной обработке строк подсказала необходимость переписать его на C. В результате была резко повышена производительность программ-клиентов `string` без каких-либо изменений в интерфейсе. То есть клиенты модуля мгновенно стали быстрее без всякой модификации для нового модуля на C. Аналогичный переход был применен к модулю `pickle`, с которым мы познакомились в главе 16 «Базы данных и постоянное хранение» – заново написанный `cPickle` сохраняет совместимость, но значительно быстрее.

Это служит хорошим уроком для разработчиков Python: модули можно сначала быстро разрабатывать на Python, а затем переносить на C, если требуется увеличить их быстродействие. Поскольку интерфейсы к модулям Python и рас-

[1] Фактически в Python 2.0 модуль `string` опять был реализован заново: теперь это интерфейс к новым *методам* строк, которые могут обрабатывать также строки Unicode. Как отмечалось, большинство функций `string` в версии 2.0 доступно также в виде методов объектов. Например, `string.split(X)` теперь просто служит синонимом для `X.split()`; поддерживаются оба формата, но последний со временем может стать превалирующим. В любом случае на клиентов первоначального модуля `string` это изменение никак не повлияло – еще один урок!

ширениям C идентичны (те и другие импортируются), трансляции модулей на C обратно совместимы со своими прототипами Python. Для клиентов единственным результатом трансляции таких модулей является повышение производительности.

Обычно нет нужды переписывать все модули на C перед поставкой клиенту приложения: можно отобрать те модули, которые являются критическими для быстродействия (такие как `string` и `pickle`), и транслировать их, а остальные оставить на Python. Для выявления модулей, трансляция которых на C может дать наибольший выигрыш, используйте технику хронометража и профилирования, описанную в предыдущей главе. Модули расширения, написанные на C, будут представлены в следующей части книги.

Подробнее об оболочке экспертной системы holmes

Как фактически можно применять эти правила? Как упоминалось, анализатор правил, с которым мы только что познакомились, входит в состав оболочки экспертной системы holmes, написанной на Python. Из-за недостатка места holmes не освещается в этой книге подробно; ее код и документацию можно найти в каталоге *PP2E\AI\ExpertSystem* на прилагаемом к книге CD. Но в порядке знакомства скажем, что holmes является машиной логического вывода, осуществляющей прямой и обратный логический вывод согласно задаваемым правилам. Например, правило:

```
rule pylike if ?X likes coding, ?X likes spam then ?X likes Python
```

можно использовать для проверки того, что некто любит Python (обратный, от «then» к «if»), и для вывода о том, что некто любит Python, из множества известных фактов (прямой, от «if» к «then»). Для логического вывода может использоваться несколько правил, а правила, указывающие на одинаковое заключение, представляют альтернативы. Holmes также осуществляет попутно простой поиск по шаблону для присваивания значений переменным, имеющимся в правилах (например, `?X`), и может объяснить свои действия.

Чтобы это все стало более конкретным, пройдем небольшой сеанс работы с holmes. Интерактивная команда `+=` добавляет в базу новое правило, выполняя анализатор правил, а `@@` выводит текущую базу правил:

```
C:_..\PP2E\Ai\ExpertSystem\holmes\holmes>python holmes.py
-Holmes inference engine-
holmes> += rule pylike if ?X likes coding, ?X likes spam then ?X likes Python
holmes> @@
rule pylike if ?X likes coding, ?X likes spam then ?X likes Python.
```

Теперь, чтобы начать построение обратной цепочки доказательства, используется команда `?-`. Здесь приведено объяснение доказательства; holmes может также сообщить, почему он задает вопрос. Переменные шаблонов holmes могут появляться как в правилах, так и в запросах; в правилах переменные обеспечивают обобщение; в запросе они обеспечивают ответ:

```
holmes> ?- mel likes Python
is this true: "mel likes coding" ? y
is this true: "mel likes spam" ? y
yes: (no variables)

show proof ? yes
```

```
  "mel likes Python" by rule pylike
      "mel likes coding" by your answer
      "mel likes spam" by your answer
more solutions? n

holmes> ?- linda likes ?X
is this true: "linda likes coding" ? y
is this true: "linda likes spam" ? y
yes: linda likes Python
```

Построение прямой цепочки рассуждений от группы фактов к заключению начинается командой +-. Здесь применяется то же правило, но иным способом:

```
holmes> +- chris likes spam, chris likes coding
I deduced these facts...
    chris likes Python
I started with these facts...
    chris likes spam
    chris likes coding
time: 0.0
```

Интересно, что выводы связываются в цепочку с помощью нескольких правил, когда часть «if» некоторого правила указана в «then» другого правила:

```
holmes> += rule 1 if thinks ?x then human ?x
holmes> += rule 2 if human ?x then mortal ?x
holmes> ?- mortal bob
is this true: "thinks bob" ? y
yes: (no variables)

holmes> +- thinks bob
I deduced these facts...
    human bob
    mortal bob
I started with these facts...
    thinks bob
time: 0.0
```

Наконец, команда @= используется для загрузки файлов правил, которые реализуют более сложные базы знаний; анализатор правил применяется к каждому правилу в файле. Вот файл, в котором закодированы правила классификации животных; если хотите поэкспериментировать, то на CD есть другие файлы примеров:

```
holmes> @= ..\bases\zoo.kb
holmes> ?- it is a penguin
is this true: "has feathers" ? why
to prove "it is a penguin" by rule 17
this was part of your original query.
is this true: "has feathers" ? y
is this true: "able to fly" ? n
is this true: "black color" ? y
yes: (no variables)
```

Чтобы завершить сеанс, введите «stop», а чтобы получить полный список команд, введите «help». Дополнительные сведения есть в текстовых файлах в каталогах holmes. Holmes – старая система, написанная до Python 1.0 (около 1993 года), но работает без изменений на всех платформах под Python 1.5.2.

Поиск регулярных выражений

Разбиение и соединение строк представляют собой простой способ обработки текста, если только он соответствует предполагаемому формату. Для более общих задач анализа текста в Python есть средства поиска, использующие регулярные выражения. Регулярные выражения (RE) являются строками, определяющими *образцы*, которые должны сравниваться с другими строками. Вы указываете образец и строку и спрашиваете, соответствует ли строка вашему образцу. После нахождения соответствия части строки, соответствующие частям образца, становятся доступными сценарию. Таким образом, поиск соответствия не только дает ответ «да/нет», но и позволяет выбирать подстроки.

Строки образцов регулярных выражений могут быть сложными (скажем честно – они могут выглядеть чудовищно). Но освоившись с ними, можно использовать их вместо больших кодируемых вручную подпрограмм для поиска в строках. В Python регулярные выражения не входят в состав синтаксиса самого языка, но поддерживаются модулями расширения, которые нужно импортировать. Модули определяют функции для компиляции строк образцов в объекты образцов, сопоставления этих объектов со строками и выбора совпавших подстрок после найденного соответствия.

Помимо этих общих положений нужно отметить, что история использования регулярных выражений в Python не совсем проста:

Модуль regex (старый)

> В ранних версиях Python стандартным (и единственным) модулем RE был `regex`. Он был быстрым и поддерживал образцы, написанные в стиле *awg*, *grep* и *emacs*, но сейчас несколько устарел (хотя, вероятно, по-прежнему будет доступен в течение еще некоторого времени).

Модуль re (новый)

> Сегодня следует использовать `re`, новый модуль RE для Python, который появился примерно вместе с Python версии 1.5. Этот модуль обеспечивает значительно более богатый синтаксис образцов RE, который стремится приблизиться к способу кодирования образцов в языке Perl (да, RE является особенностью Perl, достойной эмуляции). Например, `re` поддерживает понятия именованных групп, классы символов и *нежадный (non-greedy)* поиск – операторы образцов RE, соответствующие возможно меньшему числу символов (другие операторы образцов RE всегда соответствуют самой длинной возможной подстроке).

До самого недавнего времени `re` в целом был медленнее `regex`, поэтому приходилось выбирать между скоростью и сходным с Perl синтаксисом RE. Однако сегодня `re` оптимизирован в реализации `sre` в такой мере, что `regex` больше не предлагает каких-либо очевидных преимуществ. Более того, `re` в Python 2.0 теперь поддерживает поиск строк Unicode (строк с 16-разрядными символами для представления больших наборов символов).

Из-за этого перехода я переписал примеры RE в этой книге, чтобы использовать в них новый модуль `re` вместо `regex`. Старые, основанные на `regex` версии все еще имеются на прилагаемом к книге CD в каталоге *PP2E\lang\old-regex*. Если окажется, что вы вынуждены перейти на старый код `regex`, можете также найти документ, описывающий необходимые для трансляции шаги, на *http://www.python.org*. Интерфейсы обоих модулей аналогичны, но в `re` появился объект поиска соответствия и незначительно изменен синтаксис образцов.

Сказав это, я хочу также предупредить, что RE является сложной темой, которую невозможно глубоко здесь осветить. Если эта область вызвала ваш интерес, то вам будет полезна книга «*Mastering Regular Expressions*» издательства O'Reilly.

Использование модуля re

Модуль Python re содержит функции, которые могут искать образцы сразу или создавать компилированные *объекты образцов* для выполнения поиска в будущем. Объекты образцов (и вызовы поиска модуля) в свою очередь генерируют *объекты соответствия*, которые содержат информацию об успешных соответствиях и соответствующих подстроках. Следующие несколько разделов описывают интерфейсы модуля и некоторые операторы, которые можно использовать для написания образцов.

Функции модуля

Верхний уровень модуля предоставляет функции для поиска, замены, предварительной компиляции и так далее:

`compile(pattern [, flags])`

Компилировать строку образца PB `pattern` в объект регулярного выражения для поиска в будущем. Значение аргумента флагов смотрите в справочном руководстве.

`match(pattern, string [, flags])`

Если ноль или более символов в начале `string` соответствуют строке образца `pattern`, возвращает соответствующий экземпляр `MatchObject`, или `None`, если соответствие не найдено.

`search(pattern, string [, flags])`

Искать в `string` место, соответствующее образцу `pattern`, и возвратить соответствующий экземпляр `MatchObject` либо `None`, если соответствие не найдено.

`split(pattern, string [, maxsplit])`

Разбить `string` по вхождениям `pattern`. Если в образце используются захватывающие `()`, то возвращаются также вхождения образцов или подобразцов.

`sub(pattern, repl, string [, count])`

Возвратить строку, полученную заменой (первых `count`) самых левых неперекрывающихся вхождений `pattern` (строки или объекта RE) в `string` на `repl`.

`subn(pattern, repl, string [, count])`

То же, что sub, но возвращает кортеж: (`новая строка`, `число сделанных изменений`).

Компилированные объекты образцов

На следующем уровне объекты образцов предоставляют аналогичные атрибуты, но строка образца подразумевается. Функция `re.compile` в предыдущем разделе полезна для оптимизации образцов, которые могут отыскиваться неоднократно (компилированные образцы ищутся быстрее). У объектов образцов, возвращаемых `re.compile`, есть такие атрибуты:

`match(string [, pos] [, endpos])`

`search(string [, pos] [, endpos])`

`split(string [, maxsplit])`

`sub(repl, string [, count])`

`subn(repl, string [, count])`

То же, что функции `re`, но образец подразумевается, а `pos` и `endpos` указывают позицию начала/конца поиска соответствия в строке.

Объекты соответствия

Наконец, если функция или метод `match` или `search` выполнены успешно, возвращается объект соответствия (`None`, если поиск соответствия безуспешен). Объекты соответствия экспортируют ряд собственных атрибутов, в том числе:

`group([g1, g2, ...])`

Возвращает указанные подстроки, соответствующие группам, определяемым круглыми скобками в образце.

`groups()`

Возвращает кортеж всех подстрок, соответствующих группам.

`start([group]), end([group])`

Индексы начала и конца подстроки, соответствующей `group` (или всей найденной строки, если нет `group`).

`span([group])`

Возвращает кортеж из двух элементов: `(start(group), end(group))`.

Образцы регулярных выражений

Строки регулярных выражений строятся путем сцепления односимвольных форматов регулярных выражений, представленных в табл. 18.1. Обычно для каждого формата ищется самая длинная соответствующая ему строка, за исключением нежадных операторов. В таблице `R` означает любое регулярное выражение, `C` обозначает символ, а `N` обозначает цифру.

Таблица 18.1. Синтаксис образцов re

`.`	Соответствует любому символу (включая перевод строки, если задан флаг DOTALL)
`^`	Соответствует началу строки (каждой строчки в режиме MULTILINE)
`$`	Соответствует концу строки (каждой строчки в режиме MULTILINE)
`C`	Любой неспециальный символ соответствует самому себе
`R*`	Ноль или более предшествующих регулярных выражений `R` (как можно больше)
`R+`	Одно или более предшествующих регулярных выражений `R` (как можно больше)
`R?`	Ноль или одно вхождение предшествующего регулярного выражения `R`
`R{m,n}`	Соответствует от `m` до `n` повторениям предшествующего регулярного выражения `R`
`R*?, R+?, R??, R{m,n}?`	То же, что `*`, `+` и `?`, но соответствует минимально возможному числу повторений; известны как *нежадные* операторы (в отличие от прочих, ищут и используют как можно меньше символов)
`[ ]`	Определяет набор символов: например, `[a-zA-Z]` соответствует любой букве
`[^ ]`	Определяет дополнительный набор символов: соответствует, если символа нет в наборе
`\`	Преобразует специальные символы (например, `*?+|()`) и вводит специальные последовательности

Таблица 18.1 (продолжение)

`\\`	Литерально соответствует `\` (пишется в образце как `\\\\` или `r\\`)
`R|R`	Альтернатива: соответствие левому или правому `R`
`RR`	Конкатенация: соответствие обоим `R`
`(R)`	Соответствует любому R внутри `()` и создает группу (сохраняет найденную подстроку)
`(?: R)`	То же, но не создает группу
`(?= R)`	Совпадение, если `R` соответствует следующим символам строки, но не поглощает их (например, `X (?=Y)` соответствует `X`, только если за ним идет `Y`)
`(?! R)`	Соответствует, если `R` не соответствует следующим. Обратно для `(?=R)`
`(?P<name> R)`	Соответствует любому `R` в `()` и создает именованную группу
`(?P=name)`	Соответствует тексту, найденному предшествующей группой с именем `name`
`(?#...)`	Комментарий; игнорируется
`(?letter)`	Флаг режима; буква может иметь значение `i`, `L`, `m`, `s`, `x` (см. руководство)

В образцах можно сочетать диапазоны и символы. Например, `[a-zA-Z0-9_]+` соответствует самой длинной строке из одного или более символов букв, цифр или подчеркиваний. Специальные символы преобразуются, как обычно в строках Python: `[\t ]*` соответствует нулю или более табуляций и пробелов (то есть пропускаются пробельные символы).

Конструкция группирования с помощью круглых скобок, `(R)`, позволяет извлекать совпавшие подстроки после успешного поиска соответствия. Часть строки, соответствующая выражению в скобках, сохраняется в нумерованном регистре. Доступ к ней после успешного поиска производится с помощью метода `group` объекта соответствия.

Помимо описанного в этой таблице, в образцах можно также использовать специальные последовательности, представленные в табл. 18.2. В соответствии с правилами Python для строк иногда нужно удваивать обратную косую черту (`\\`) или пользоваться необработанными строками Python (`r'...'`), чтобы сохранить обратные слэши в образце.

Таблица 18.2. Специальные последовательности re

`\num`	Совпадение с текстом группы `num` (нумеруются с 1)
`\A`	Соответствует только в начале строки
`\b`	Пустая строка на границе слов
`\B`	Пустая строка не на границе слов
`\d`	Любая десятичная цифра (`[0-9]`)
`\D`	Любой символ, отличный от десятичной цифры (как `[^0-9]`)
`\s`	Любой пробельный символ (как `[ \t\n\r\f\v]`)
`\S`	Любой непробельный символ (как `[^ \t\n\r\f\v]`)
`\w`	Любой буквенно-цифровой символ (использует флаг LOCALE)
`\W`	Любой символ, отличный от буквенно-цифрового (использует флаг LOCALE)
`\Z`	Соответствует только в конце строки

В библиотечном руководстве Python есть дополнительные детали. Но чтобы продемонстрировать типичное использование интерфейсов `re`, обратимся к нескольким коротким примерам.

Основные образцы

Чтобы проиллюстрировать, как объединяются образцы, начнем с нескольких коротких тестовых файлов, ищущих соответствие образцам простого вида. Комментарии в примере 18.3 описывают осуществляемые операции; смотрите по табл. 18.1, какие операторы здесь используются.

Пример 18.3. PP2E\lang\re-basics.py

```
# литералы, наборы, диапазоны   (все выводят 2 = смещение, где найден образец)

import re                                   # надлежит использовать сегодня

pattern, string = "A.C.", "xxABCDxx"        # неспециальные символы соответствуют
                                            # самим себе; '.' означает один любой
matchobj = re.search(pattern, string)       # символ; search возвращает объект
if matchobj:                                # соответствия или None
    print matchobj.start()                  # start дает индекс начала соответствия

pattobj  = re.compile("A.*C.*")             # 'R*' означает 0 или более R
matchobj = pattobj.search("xxABCDxx")       # compile возвращает объект образца
if matchobj:                                # patt.search возвращает объект соответствия
    print matchobj.start()

# наборы выбора
print re.search(" *A.C[DE][D-F][^G-ZE]G\t+ ?", "..ABCDEFG\t..").start()

# альтернативы
print re.search("A|XB|YC|ZD", "..AYCD..").start()  # R1|R2 означает R1 или R2

# границы слов
print re.search(r"\bABCD", "..ABCD ").start() # \b означает границу слова
print re.search(r"ABCD\b", "..ABCD ").start() # использовать r'...' для сохранения '\'
```

Обратите внимание, что есть несколько способов начать поиск соответствия с помощью `re` – при вызове функций поиска модуля и создавая компилированные объекты образца. В любом случае, можно использовать или нет получаемый объект соответствия. Все операторы print этого сценария выводят результат 2 – смещение, по которому образец был найден в строке. Например, в первом тесте «A.C.» соответствует «ABCD» со смещением 2 в проверяемой строке (то есть после начальных «xx»):

```
C:\...\PP2E\Lang>python re-basic.py
2
2
2
2
2
2
```

В примере 18.4 части строк образцов, заключенные в круглые скобки, выделяют *группы*; части строки, которым они соответствуют, доступны после поиска.

Пример 18.4. PP2E\lang\re-groups.py

```
# группы (извлечь подстроки, соответствующие частям образцов в '()')

import re
```

```
patt = re.compile("A(.)B(.)C(.)")                    # сохраняет 3 подстроки
mobj = patt.match("A0B1C2")                          # каждая '()' является группой, 1..n
print mobj.group(1), mobj.group(2), mobj.group(3)    # group() возвращает подстроку
patt = re.compile("A(.*)B(.*)C(.*)")                 # сохраняет 3 подстроки
mobj = patt.match("A000B111C222")                    # groups() дает все группы
print mobj.groups()

print re.search("(A|X)(B|Y)(C|Z)D", "..AYCD..").groups()

patt = re.compile(r"[\t ]*#\s*define\s*([a-z0-9_]*)\s*(.*)")
mobj = patt.search(" # define  spam  1 + 2 + 3")     # части #define в C
print mobj.groups()                                  # \s - пробельный символ
```

Например, в первом тесте есть три группы (.), каждая из которых соответствует одному символу и сохраняет найденный символ; при вызове group получаем найденные совпадения. Во втором тесте группы (.*) соответствуют любому числу символов и сохраняют их. Последняя проверка здесь соответствует строкам C #define; подробнее об этом ниже.

```
C:\...\PP2E\Lang>python re-groups.py
0 1 2
('000', '111', '222')
('A', 'Y', 'C')
('spam', '1 + 2 + 3')
```

Наконец, кроме поиска соответствия и извлечения подстрок в re есть средства замены строк (см. пример 18.5).

Пример 18.5. PP2E\lang\re-subst.py

```
# подстановка (замена входящих в строку patt на repl)

import re
print re.sub('[ABC]', '*', 'XAXAXBXBXCXC')
print re.sub('[ABC]_', '*', 'XA-XA_XB-XB_XC-XC_')
```

В первом тесте заменяются все символы из набора; во втором только те, за которыми следует символ подчеркивания:

```
C:\...\PP2E\Lang>python re-subst.py
X*X*X*X*X*X*X*
XA-X*XB-X*XC-X*
```

Поиск образцов в файлах заголовков C

Сценарий примера 18.6 показывает более практическое применение этих операторов образца. С помощью регулярных выражений он находит строки #define и #include в файлах заголовков C и извлекает их составляющие. Общность этих образцов позволяет обнаруживать с их помощью целый ряд форматов строк; группы в образцах (части, заключенные в круглые скобки) используются для извлечения из строчек соответствующих подстрок после нахождения соответствия.

Пример 18.6. PP2E\Lang\cheader.py

```
#! /usr/local/bin/python
import sys, re
from string import strip

pattDefine = re.compile(                                  # компиляция в объект образца
    '^#[\t ]*define[\t ]+([a-zA-Z0-9_]+)[\t ]*(.*)')       # "# define xxx yyy..."
```

```
pattInclude = re.compile(
    '^#[\t ]*include[\t ]+[<"]([a-zA-Z0-9_/\.]+)')       # "# include <xxx>..."

def scan(file):
    count = 0
    while 1:                                             # построчный просмотр
        line = file.readline()
        if not line: break
        count = count + 1
        matchobj = pattDefine.match(line)                # None, если поиск неудачен
        if matchobj:
            name = matchobj.group(1)                     # подстроки для участков (...)
            body = matchobj.group(2)
            print count, 'defined', name, '=', strip(body)
            continue
        matchobj = pattInclude.match(line)
        if matchobj:
            start, stop = matchobj.span(1)               # индексы начала/конца (...)
            filename = line[start:stop]                  # срез строки
            print count, 'include', filename             # то же, что matchobj.group(1)

if len(sys.argv) == 1:
    scan(sys.stdin)                                      # без аргументов: чтение stdin
else:
    scan(open(sys.argv[1], 'r'))                         # аргумент: имя входного файла
```

Для проверки запустим этот сценарий с текстовым файлом из примера 18.7.

Пример 18.7. PP2E\Lang\test.h

```
#ifndef TEST_H
#define TEST_H

#include <stdio.h>
#include <lib/spam.h>
#  include   "Python.h"

#define DEBUG
#define HELLO 'hello regex world'
#  define SPAM    1234

#define EGGS sunny + side + up
#define  ADDER(arg) 123 + arg
#endif
```

Обратите внимание на пробелы после # в некоторых из этих строк; регулярные выражения достаточно гибки, чтобы учитывать такие отклонения от нормы. Вот как работает этот сценарий, выбирая строки `#include` и `#define` и их части; для каждой найденной строчки выводится ее номер, тип и найденные подстроки:

```
C:\...\PP2E\Lang>python cheader.py test.h
2 defined TEST_H =
4 include stdio.h
5 include lib/spam.h
6 include Python.h
8 defined DEBUG =
9 defined HELLO = 'hello regex world'
10 defined SPAM = 1234
12 defined EGGS = sunny + side + up
13 defined ADDER = (arg) 123 + arg
```

Утилита для поиска в файлах по образцу

В следующем сценарии осуществляется поиск образца в группе файлов, подобно программе командной строки `grep`. Мы уже писали программы поиска в файлах и каталогах в главе 5 «Более крупные системные примеры, часть 2». Теперь в файлах будут отыскиваться образцы, а не просто строки (см. пример 18.8). Образцы вводятся интерактивно, разделяясь пробелами, а файлы, в которых должен осуществляться поиск, задаются по образцу ввода для инструмента Python расширения имен файлов `glob.glob`, который мы тоже уже изучали.

Пример 18.8. PP2E\Lang\pygrep1.py

```
#!/usr/local/bin/python
import sys, re, glob
from string import split

help_string = """
Usage options.
interactive:  % pygrep1.py
"""

def getargs():
    if len(sys.argv) == 1:
        return split(raw_input("patterns? >")), raw_input("files? >")
    else:
        try:
            return sys.argv[1], sys.argv[2]
        except:
            print help_string
            sys.exit(1)

def compile_patterns(patterns):
    res = []
    for pattstr in patterns:
        try:
            res.append(re.compile(pattstr))            # создать объект образца re
        except:                                        # или использовать re.match
            print 'pattern ignored:', pattstr
    return res

def searcher(pattfile, srchfiles):
    patts = compile_patterns(pattfile)                 # компилировать для скорости
    for file in glob.glob(srchfiles):                  # все подходящие файлы
        lineno = 1                                     # glob тоже использует re
        print '\n[%s]' % file
        for line in open(file, 'r').readlines():       # все строки файла
            for patt in patts:
                if patt.search(line):                  # опробовать все образцы
                    print '%04d)' % lineno, line,      # соответствие, если не None
                    break
            lineno = lineno+1

if __name__== '__main__':
    apply(searcher, getargs())
```

Вот результат типичного прогона этого сценария; производится поиск во всех файлах Python текущего каталога двух разных образцов, для быстроты откомпилированных. Обратите внимание, что имена файлов тоже заданы образцом – модуль Python `glob` внутренне использует `re`:

```
C:\...\PP2E\Lang>python pygrep1.py
```

```
patterns? >import.*string spam
files? >*.py

[cheader.py]

[finder2.py]
0002) import string, glob, os, sys

[patterns.py]
0048) mobj = patt.search(¨ # define  spam  1 + 2 + 3¨)

[pygrep1.py]

[rules.py]

[summer.py]
0002) import string

[__init__.py]
```

Генераторы парсеров

Если у вас есть некоторая подготовка в теории синтаксического анализа, то вы должны знать, что ни регулярных выражений, ни разбиения строк недостаточно для работы с более сложными грамматиками языков (грубо говоря, у них нет «памяти», необходимой настоящим грамматикам). Для более сложных задач анализа языков иногда требуется развитый парсер (программа синтаксического анализа). Поскольку Python позволяет интегрировать средства C, можно осуществлять его интеграцию с традиционными системами генерации парсеров, такими как *yacc* и *bison*. Еще лучше воспользоваться уже существующими средствами интеграции.

Есть также системы синтаксического анализа для Python, которые можно взять с сайта Python. В их числе система *kwParsing*, разработанная Ароном Уотерсом (Aaron Watters), представляющая собой генератор парсеров, написанный на Python, и инструментальный набор *SPARK*, разработанный Джоном Эйкоком (John Aycock) и представляющий собой облегченную систему, применяющую алгоритм Earley для решения технических проблем, связанных с генерацией парсеров LALR (если вы не знаете, что это такое, то вам, вероятно не стоит и беспокоиться). Однако поскольку все это сложные инструменты, мы пропустим их детали в этой книге. Обратитесь на *http://www.python.org* за информацией о средствах генераторов парсеров, которые можно использовать в программах Python.

Особый интерес для данной главы представляет YAPPS – Yet Another Python Parser System (еще одна система синтаксического анализа Python). YAPPS – это генератор парсеров, написанный на Python. С помощью заданных правил он генерирует код Python в доступном для чтения виде, в котором реализуется парсер, использующий метод рекурсивного спуска. Генерируемые YAPPS парсеры весьма похожи на закодированные вручную парсеры выражений, представленные в следующем разделе (которыми они и инспирированы). YAPPS создает парсеры LL(1) не такие мощные, как парсеры LALR, но достаточные для многих задач языков. Подробнее о YAPPS смотрите *http://theory.standford.edu/~amitp/Yapps*.

Парсеры, написанные вручную

Так как Python является универсальным языком, можно попробовать написать на нем парсер вручную. Например, синтаксический анализ методом рекурсивного спуска (*recursive descent parsing*) – довольно хорошо известная техника анализа языковой

Урок 2: не нужно изобретать колесо

По поводу генераторов парсеров: для использования некоторых из этих инструментов в программах Python вам потребуется интегрирующий их модуль расширения. В таких ситуациях первым делом нужно проверить, не существует ли уже такого расширения в общественном владении. В особенности для таких стандартных инструментов весьма вероятно, что кто-то уже написал интегрирующий модуль, который можно использовать в готовом виде, а не писать с самого начала новый.

Конечно, не каждый может подарить все свои модули расширения обществу, но библиотека имеющихся компонентов, которые можно брать бесплатно, растет, и есть сообщество специалистов, к которым можно обратиться с вопросом. Ссылки на программные ресурсы Python можно найти на *http://www.python.org*. При наличии во всем мире на момент написания книги полумиллиона пользователей Python в разделе прототипов можно найти немало.

информации. Поскольку Python – язык очень высокого уровня, написать на нем собственно парсер обычно проще, чем на традиционных языках типа C или C++.

Для иллюстрации в этом разделе будет разработан специальный парсер для простой грамматики: он разбирает и вычисляет строки арифметических выражений. Этот пример также демонстрирует возможности Python как универсального языка программирования. Хотя Python часто используется в качестве интерфейса или языка быстрой разработки приложений, он полезен и для таких вещей, которые обычно пишутся на языках системных разработок типа C или C++.

Грамматика выражения

Грамматику, которую будет распознавать наш парсер, можно описать так:

```
goal -> <expr> END                        [number, variable, ( ]
goal -> <assign> END                      [set]

assign -> 'set' <variable> <expr>         [set]

expr -> <factor> <expr-tail>              [number, variable, ( ]

expr-tail -> ^                            [END, ) ]
expr-tail -> '+' <factor> <expr-tail>     [+]
expr-tail -> '-' <factor> <expr-tail>     [-]

factor -> <term> <factor-tail>            [number, variable, ( ]

factor-tail -> ^                          [+, -, END, ) ]
factor-tail -> '*' <term> <factor-tail>   [*]
factor-tail -> '/' <term> <factor-tail>   [/]

term -> <number>                          [number]
term -> <variable>                        [variable]
term -> '(' <expr> ')'                    [(]

tokens: (, ), num, var, -, +, /, *, set, end
```

Это довольно типичная грамматика для простого языка выражений, допускающая произвольную вложенность выражений (некоторые образцы выражений есть в конце листинга модуля `testparser` в примере 18.11). Анализируемые строки представляют собой выражение или присваивание переменной (`set`). В выражениях участвуют чис-

ла, переменные и операторы +, -, * и /. Поскольку в грамматике factor вложен в expr, то * и / имеют более высокий приоритет (то есть связывают сильнее), чем + и -. Выражения можно заключать в круглые скобки, чтобы отменять старшинство операций, и все операторы являются ассоциативными слева (например, 1-2-3 рассматривается как (1-2)-3).

Лексемы (*tokens*) представляют собой наиболее элементарные составляющие языка выражений. За каждым приведенным выше грамматическим правилом в квадратных скобках следует список лексем, по которым оно выбирается. При рекурсивном нисходящем анализе мы определяем набор лексем, которые могут начинать подстроку правила, и с помощью этой информации заранее предсказываем правило, которое будет работать. Для правил, которые повторяются (правил -tail), используется набор возможных последующих лексем, чтобы знать, когда остановиться. Обычно лексемы распознаются обработчиком строк (лексическим анализатором, или «сканером»), а обработчик более высокого уровня (синтаксический анализатор, или «парсер») использует поток лексем для предсказания и прохода грамматических правил и подстрок.

Код парсера

Система организована в виде двух модулей, содержащих два класса:

- *Сканер* производит посимвольный анализ нижнего уровня.
- *Парсер* содержит в себе сканер и производит грамматический анализ высокого уровня.

Парсер отвечает также за вычисление значения выражения и тестирование системы. В данной версии парсер вычисляет выражение во время его грамматического разбора. Чтобы использовать систему, с помощью входной строки создается парсер и вызывается его метод parse. Впоследствии можно снова вызывать parse с новой строкой выражения.

Здесь намеренно произведено разделение труда. Сканер извлекает из строки лексемы, но ему ничего не известно о грамматике. Парсер обрабатывает грамматику, но находится в неведении относительно самой строки. Благодаря такой модульной структуре код сохраняет относительную простоту. И это еще один пример отношений композиции ООП в действии: парсеры содержат сканеры и делегируют им полномочия.

Модуль примера 18.9 выполняет задачу лексического анализа – обнаружения в выражении базовых лексем путем сканирования по требованию строки текста слева направо. Обратите внимание на простоту используемой здесь логики; такой анализ иногда можно выполнить с помощью регулярных выражений (описанных раньше), но, на мой взгляд, образец, необходимый для обнаружения и извлечения лексем в этом примере, окажется слишком сложным и хрупким. Если вам так не кажется, попробуйте переписать этот модуль с помощью re.

Пример 18.9. PP2E\Lang\Parser\scanner.py

```
###########################################################
# сканер (лексический анализатор)
###########################################################

import string
SyntaxError    = 'SyntaxError'          # локальные ошибки
LexicalError   = 'LexicalError'

class Scanner:
    def __init__(self, text):
        self.next = 0
```

```
        self.text = text + '\0'

    def newtext(self, text):
        Scanner.__init__(self, text)

    def showerror(self):
        print '=> ', self.text
        print '=> ', (' ' * self.start) + '^'

    def match(self, token):
        if self.token != token:
            raise SyntaxError, [token]
        else:
            value = self.value
            if self.token != '\0':
                self.scan()                  # очередные лексема/значение
            return value                     # возврат предыдущего значения

    def scan(self):
        self.value = None
        ix = self.next
        while self.text[ix] in string.whitespace:
            ix = ix+1
        self.start = ix

        if self.text[ix] in ['(', ')', '-', '+', '/', '*', '\0']:
            self.token = self.text[ix]
            ix = ix+1

        elif self.text[ix] in string.digits:
            str = ''
            while self.text[ix] in string.digits:
               str = str + self.text[ix]
               ix = ix+1
            if self.text[ix] == '.':
                str = str + '.'
                ix = ix+1
                while self.text[ix] in string.digits:
                   str = str + self.text[ix]
                   ix = ix+1
                self.token = 'num'
                self.value = string.atof(str)
            else:
                self.token = 'num'
                self.value = string.atol(str)

        elif self.text[ix] in string.letters:
            str = ''
            while self.text[ix] in (string.digits + string.letters):
                str = str + self.text[ix]
                ix = ix+1
            if string.lower(str) == 'set':
                self.token = 'set'
            else:
                self.token = 'var'
                self.value = str

        else:
            raise LexicalError
        self.next = ix
```

Класс модуля парсера создает и встраивает сканер для выполнения задач лексического анализа, занимается интерпретацией грамматических правил и вычислением результата выражения, как показано в примере 18.10.

Пример 18.10. PP2E\Lang\Parser\parser1.py

```
##########################################################
# парсер (анализатор синтаксиса, вычисляет во время анализа)
##########################################################

UndefinedError = 'UndefinedError'
from scanner import Scanner, LexicalError, SyntaxError

class Parser:
    def __init__(self, text=''):
        self.lex  = Scanner(text)                    # встроить сканер
        self.vars = {'pi':3.14159}                   # добавить переменную

    def parse(self, *text):
        if text:                                     # главная точка входа
            self.lex.newtext(text[0])                # использовать парсер повторно?
        try:
            self.lex.scan()                          # получить первую лексему
            self.Goal()                              # разобрать предложение
        except SyntaxError:
            print 'Syntax Error at column:', self.lex.start
            self.lex.showerror()
        except LexicalError:
            print 'Lexical Error at column:', self.lex.start
            self.lex.showerror()
        except UndefinedError, name:
            print "'%s' is undefined at column:" % name, self.lex.start
            self.lex.showerror()

    def Goal(self):
        if self.lex.token in ['num', 'var', '(']:
            val = self.Expr()
            self.lex.match('\0')                     # выражение?
            print val
        elif self.lex.token == 'set':                # команда set?
            self.Assign()
            self.lex.match('\0')
        else:
            raise SyntaxError

    def Assign(self):
        self.lex.match('set')
        var = self.lex.match('var')
        val = self.Expr()
        self.vars[var] = val                         # присваивание имени в словаре

    def Expr(self):
        left = self.Factor()
        while 1:
            if self.lex.token in ['\0', ')']:
                return left
            elif self.lex.token == '+':
                self.lex.scan()
                left = left + self.Factor()
            elif self.lex.token == '-':
```

```
                self.lex.scan()
                left = left - self.Factor()
            else:
                raise SyntaxError

    def Factor(self):
        left = self.Term()
        while 1:
            if self.lex.token in ['+', '-', '\0', ')']:
                return left
            elif self.lex.token == '*':
                self.lex.scan()
                left = left * self.Term()
            elif self.lex.token == '/':
                self.lex.scan()
                left = left / self.Term()
            else:
                raise SyntaxError

    def Term(self):
        if self.lex.token == 'num':
            val = self.lex.match('num')                # числа
            return val
        elif self.lex.token == 'var':
            if self.vars.has_key(self.lex.value):
                val = self.vars[self.lex.value]        # найти значение для имени
                self.lex.scan()
                return val
            else:
                raise UndefinedError, self.lex.value
        elif self.lex.token == '(':
            self.lex.scan()
            val = self.Expr()                          # подвыражение
            self.lex.match(')')
            return val
        else:
            raise SyntaxError

if __name__== '__main__':
    import testparser                                  # код самотестирования
    testparser.test(Parser, 'parser1')                 # тест локального Parser
```

Если внимательно изучить этот код, то можно заметить, что парсер ведет словарь (`self.vars`) имен переменных: они сохраняются в словаре по команде *set* и выбираются из него при появлении в выражении. Лексемы представляются строками с возможными ассоциированными значениями (числовое значение для чисел и строки для имен переменных).

В этом парсере используется итерация (циклы `while`) вместо рекурсии для правил `expr-tail` и `factor-tail`. За исключением этой оптимизации, правила грамматики непосредственно отображаются в методы парсера: лексемы становятся обращениями к сканеру, а ссылки на вложенные правила становятся обращениями к другим методам.

При выполнении файла *parser1.py* как программы верхнего уровня выполняется его код самотестирования, который, в свою очередь, просто выполняет готовый тест, показанный в примере 18.11. Обратите внимание, что в вычислениях с целыми числами используются длинные целые Python (с неограниченной точностью), потому что сканер преобразует числа в строки с помощью `string.atol`. Заметьте также, что смешанные операции над целыми числами и числами с плавающей точкой приводятся к чис-

лам с плавающей точкой, так как для фактических вычислений используются операторы Python.

Пример 18.11. PP2E\Lang\Parser\testparser.py

```
######################################################
# код теста парсера
######################################################

def test(ParserClass, msg):
    print msg, ParserClass
    x = ParserClass('4 ю 2 + 3')             # допускает разные Parser
    x.parse()

    x.parse('3 + 4 / 2')                     # как eval('3 + 4 / 2')...
    x.parse('(3 + 4) / 2')
    x.parse('4 / (2 + 3)')
    x.parse('4.0 / (2 + 3)')
    x.parse('4 / (2.0 + 3)')
    x.parse('4.0 / 2 * 3')
    x.parse('(4.0 / 2) * 3')
    x.parse('4.0 / (2 * 3)')
    x.parse('(((3))) + 1')

    y = ParserClass()
    y.parse('set a 4 / 2 + 1')
    y.parse('a * 3')
    y.parse('set b 12 / a')
    y.parse('b')

    z = ParserClass()
    z.parse('set a 99')
    z.parse('set a a + 1')
    z.parse('a')

    z = ParserClass()
    z.parse('pi')
    z.parse('2 * pi')
    z.parse('1.234 + 2.1')

def interact(ParserClass):                     # ввод из командной строки
    print ParserClass
    x = ParserClass()
    while 1:
        cmd = raw_input('Enter=> ')
        if cmd == 'stop':
            break
        x.parse(cmd)
```

Сопоставьте следующие результаты с командами print в модуле самотестирования:

```
C:\...\PP2E\Lang\Parser>python parser1.py
parser1 __main__.Parser
5L
5L
3L
0L
0.8
0.8
6.0
6.0
0.666666666667
```

```
4L
9L
4L
100L
3.14159
6.28318
3.334
```

Как обычно, можно тестировать и использовать систему интерактивно:

```
% python
>>> import parser1
>>> x = parser1.Parser()
>>> x.parse('1 + 2')
3L
```

Ошибки перехватываются и комментируются:

```
>>> x.parse('1 + a')
'a' is undefined at column: 4
=>  1 + a
=>      ^
>>> x.parse('1+a+2')
'a' is undefined at column: 2
=>  1+a+2
=>    ^
>>> x.parse('1 * 2 $')
Lexical Error at column: 6
=>  1 * 2 $
=>        ^
>>> x.parse('1 * - 1')
Syntax Error at column: 4
=>  1 * - 1
=>      ^
>>> x.parse('1 * (9')
Syntax Error at column: 6
=>  1 * (9
=>        ^
```

Патологически большие числа обрабатываются нормально, потому что для этого используются встроенные объекты и операторы Python:

```
>>> x.parse('88888888888888888888888888888888888888888888888.9999999')
8.88888888889e+44
>>> x.parse('99999999999999999999999999999999999999999 + 2')
100000000000000000000000000000000000000001L
>>> x.parse('99999999999999999999999999.88888888888 + 1.1')
1e+30
```

Кроме того, в модуле `testparser` есть интерфейс интерактивного цикла, чтобы использовать парсер как простой калькулятор командной строки (или когда надоест вводить вызовы методов парсера). Передайте класс `Parser`, чтобы `testparser` мог создать свой собственный:

```
>>> import testparser
>>> testparser.interact(parser1.Parser)
Enter=> 4 * 3 + 5
17L
Enter=> 5 + 4 * 3
17L
```

```
Enter=> (5 + 4) * 3
27L
Enter=> set a 99
Enter=> set b 66
Enter=> a + b
165L
Enter=> # + 1
Lexical Error at column: 0
=>  # + 1
=>  ^
Enter=> a * b + c
'c' is undefined at column: 8
=>  a * b + c
=>          ^
Enter=> a * b * + c
Syntax Error at column: 8
=>  a * b * + c
=>          ^
Enter=> a
99L
Enter=> a * a * a
970299L
Enter=> stop
>>>
```

Урок 3: разделяй и властвуй

Как демонстрирует система синтаксического анализа, модульная конструкция программы почти всегда приносит значительный выигрыш. С помощью средств структурирования Python (функций, модулей, классов и т. п.) большие задачи могут быть разбиты на маленькие контролируемые части, которые можно кодировать и тестировать независимо друг от друга.

Например, сканер можно протестировать без парсера, создав его экземпляр с помощью входной строки и повторно вызывая методы `scan` или `match`. Можно даже делать такую проверку интерактивно из командной строки Python. В результате разделения программ на логические составляющие становится проще понять их и модифицировать. Представьте себе, как бы выглядел парсер, если бы логика сканера была вложена в него, а не вызывалась.

Добавление интерпретатора дерева синтаксического разбора

Одна из слабостей программы `parser1` состоит в том, что она встраивает логику вычисления выражений в логику синтаксического анализа: результат вычисляется во время грамматического разбора строки. Это ускоряет вычисление, но может затруднить модификацию кода, особенно в больших системах. Проще говоря, можно изменить организацию программы с тем, чтобы отделить синтаксический анализ от вычисления. Вместо вычисления строки парсер может построить промежуточное представление, которое будет вычислено позднее. Дополнительным стимулом создания такого отдельного представления является возможность подвергнуть его анализу другими средствами (например, оптимизаторами, вьюерами и т. п.).

В примере 18.12 показана версия `parser1`, реализующая эту мысль. Парсер анализирует строку и строит дерево синтаксического разбора, то есть дерево экземпляров клас-

сов, которое представляет выражение и может быть вычислено на отдельной стадии. Дерево синтаксического разбора строится из классов, которые «умеют» вычислять себя: чтобы вычислить выражение, мы предложим дереву вычислить себя. Корневые узлы дерева просят своих потомков вычислить себя, а затем объединяют результаты, применяя один оператор. Фактически вычисление в этой версии является рекурсивным обходом дерева вложенных экземпляров классов, построенного парсером.

Пример 18.12. PP2E\Lang\Parser\parser2.py

```
TraceDefault   = 0
UndefinedError = "UndefinedError"
from scanner import Scanner, SyntaxError, LexicalError

#####################################################
# интерпретатор (дерево умных объектов)
#####################################################

class TreeNode:
    def validate(self, dict):                   # проверка ошибок по умолчанию
        pass
    def apply(self, dict):                      # вычислитель по умолчанию
        pass
    def trace(self, level):                     # обратная сборка по умолчанию
        print '.'*level + '<empty>'

# КОРНИ

class BinaryNode(TreeNode):
    def __init__(self, left, right):            # наследуемые методы
        self.left, self.right = left, right     # левая/правая ветви
    def validate(self, dict):
        self.left.validate(dict)                # рекурсия в ветви
        self.right.validate(dict)
    def trace(self, level):
        print '.'*level + '[' + self.label + ']'
        self.left.trace(level+3)
        self.right.trace(level+3)

class TimesNode(BinaryNode):
    label = '*'
    def apply(self, dict):
        return self.left.apply(dict) * self.right.apply(dict)

class DivideNode(BinaryNode):
    label = '/'
    def apply(self, dict):
        return self.left.apply(dict) / self.right.apply(dict)

class PlusNode(BinaryNode):
    label = '+'
    def apply(self, dict):
        return self.left.apply(dict) + self.right.apply(dict)

class MinusNode(BinaryNode):
    label = '-'
    def apply(self, dict):
        return self.left.apply(dict) - self.right.apply(dict)

# ВЕТВИ

class NumNode(TreeNode):
    def __init__(self, num):
```

```
        self.num = num                          # уже число
    def apply(self, dict):                      # проверка по умолчанию
        return self.num
    def trace(self, level):
        print '.'*level + `self.num`

class VarNode(TreeNode):
    def __init__(self, text, start):
        self.name   = text                      # имя переменной
        self.column = start                     # column для ошибок
    def validate(self, dict):
        if not dict.has_key(self.name):
            raise UndefinedError, (self.name, self.column)
    def apply(self, dict):
        return dict[self.name]                  # сначала проверить
    def assign(self, value, dict):
        dict[self.name] = value                 # локальное расширение
    def trace(self, level):
        print '.'*level + self.name

# СМЕСЬ

class AssignNode(TreeNode):
    def __init__(self, var, val):
        self.var, self.val = var, val
    def validate(self, dict):
        self.val.validate(dict)                 # не проверять переменную
    def apply(self, dict):
        self.var.assign( self.val.apply(dict), dict )
    def trace(self, level):
        print '.'*level + 'set '
        self.var.trace(level + 3)
        self.val.trace(level + 3)

######################################################
# парсер (анализ синтаксиса, построение дерева)
######################################################

class Parser:
    def __init__(self, text=''):
        self.lex     = Scanner(text)            # создать сканер
        self.vars    = {'pi':3.14159}           # добавить константы
        self.traceme = TraceDefault

    def parse(self, *text):                     # внешний интерфейс
        if text:
            self.lex.newtext(text[0])           # повторно использовать с новым текстом
        tree = self.analyse()                   # разбор строки
        if tree:
            if self.traceme:                    # дамп дерева разбора?
                print; tree.trace(0)
            if self.errorCheck(tree):           # проверить имена
                self.interpret(tree)            # вычислить дерево

    def analyse(self):
        try:
            self.lex.scan()                     # получить первую лексему
            return self.Goal()                  # построить дерево разбора
        except SyntaxError:
            print 'Syntax Error at column:', self.lex.start
            self.lex.showerror()
```

```
        except LexicalError:
            print 'Lexical Error at column:', self.lex.start
            self.lex.showerror()

    def errorCheck(self, tree):
        try:
            tree.validate(self.vars)                # проверка ошибок
            return 'ok'
        except UndefinedError, varinfo:
            print "'%s' is undefined at column: %d" % varinfo
            self.lex.start = varinfo[1]
            self.lex.showerror()                    # возвращает None

    def interpret(self, tree):
        result = tree.apply(self.vars)              # дерево вычисляет себя
        if result != None:                          # игнорировать результат 'set'
            print result

    def Goal(self):
        if self.lex.token in ['num', 'var', '(']:
            tree = self.Expr()
            self.lex.match('\0')
            return tree
        elif self.lex.token == 'set':
            tree = self.Assign()
            self.lex.match('\0')
            return tree
        else:
            raise SyntaxError

    def Assign(self):
        self.lex.match('set')
        vartree = VarNode(self.lex.value, self.lex.start)
        self.lex.match('var')
        valtree = self.Expr()
        return AssignNode(vartree, valtree)             # два поддерева

    def Expr(self):
        left = self.Factor()                            # левое поддерево
        while 1:
            if self.lex.token in ['\0', ')']:
                return left
            elif self.lex.token == '+':
                self.lex.scan()
                left = PlusNode(left, self.Factor())    # добавить корневой узел
            elif self.lex.token == '-':
                self.lex.scan()
                left = MinusNode(left, self.Factor())   # растет вправо-вверх
            else:
                raise SyntaxError

    def Factor(self):
        left = self.Term()
        while 1:
            if self.lex.token in ['+', '-', '\0', ')']:
                return left
            elif self.lex.token == '*':
                self.lex.scan()
                left = TimesNode(left, self.Term())
            elif self.lex.token == '/':
```

```
                self.lex.scan()
                left = DivideNode(left, self.Term())
            else:
                raise SyntaxError

    def Term(self):
        if self.lex.token == 'num':
            leaf = NumNode(self.lex.match('num'))
            return leaf
        elif self.lex.token == 'var':
            leaf = VarNode(self.lex.value, self.lex.start)
            self.lex.scan()
            return leaf
        elif self.lex.token == '(':
            self.lex.scan()
            tree = self.Expr()
            self.lex.match(')')
            return tree
        else:
            raise SyntaxError

######################################################
# код самотестирования: использовать парсер, тестер parser1
######################################################

if __name__== '__main__':
    import testparser
    testparser.test(Parser, 'parser2')    #  выполнять с классом Parser
```

Когда `parser2` выполняется в качестве программы верхнего уровня, получается такой же вывод кода тестирования, как для `parser1`. В действительности повторно используется тот же самый код тестирования: оба парсера передают свой объект класса парсера в `testparser.test`. А так как классы являются объектами, можно также передать эту версию парсера в интерактивный цикл `testparser`: `testparser.interact(parser2.Parser)`. Внешнее поведение нового парсера такое же, как у оригинала.

Обратите внимание, что новый парсер также повторно использует тот же самый модуль сканера. Чтобы перехватывать ошибки, возбуждаемые сканером, он также импортирует особые строки, идентифицирующие исключительные ситуации сканера. И сканер, и парсер могут возбуждать исключительные ситуации при ошибках (лексические ошибки, синтаксические ошибки и ошибки неопределенных имен). Они перехватываются на верхнем уровне парсера и завершают текущий разбор. Нет надобности устанавливать и проверять флаги состояния, чтобы завершить рекурсию. Поскольку математические действия выполняются с помощью длинных целых чисел, чисел с плавающей точкой и операторов Python, обычно не требуется перехватывать числовые ошибки переполнения или потери значимости. Но в настоящем виде парсер не обрабатывает такие ошибки, как деление на ноль: они приводят к выходу из системы парсера с дампом стека Python. Выяснение причины и исправление этого оставляются в качестве упражнения.

Структура дерева синтаксического разбора

Промежуточное представление выражения является деревом экземпляров классов, форма которого отражает порядок выполнения операторов. В этом парсере есть также логика, с помощью которой выводится листинг созданного дерева с отступами, если установлен атрибут `traceme`. Отступы указывают на вложенность поддеревьев, а бинарные операторы показывают сначала левые поддеревья. Например:

```
% python
```

```
>>> import parser2
>>> p = parser2.Parser()
>>> p.traceme = 1
>>> p.parse('5 + 4 * 2')

[+]
...5L
...[*]
......4L
......2L
13L
```

При вычислении этого дерева метод `apply` рекурсивно вычисляет поддеревья и применяет операторы корня к их результатам. Здесь `*` вычисляется раньше `+`, так как находится ниже в дереве. Метод `Factor` поглощает подстроку `*` перед возвращением правого поддерева в `Expr`:

```
>>> p.parse('5 * 4 - 2')

[-]
...[*]
......5L
......4L
...2L
18L
```

В этом примере `*` вычисляется прежде `−`. Метод `Factor` проходит подстроку выражений `*` и `/` перед возвращением результирующего левого поддерева в `Expr`:

```
>>> p.parse('1 + 3 * (2 * 3 + 4)')

[+]
...1L
...[*]
......3L
......[+]
.........[*]
............2L
............3L
.........4L
31L
```

Деревья состоят из вложенных экземпляров классов. С точки зрения ООП, это еще один способ применения композиции. Так как узлы дерева являются просто экземплярами классов, это дерево можно создать и вычислить вручную:

```
PlusNode( NumNode(1),
          TimesNode( NumNode(3),
                     PlusNode( TimesNode(NumNode(2), NumNode(3)),
                               NumNode(4) ))).apply({})
```

Но можно также позволить парсеру построить его (Python не настолько похож на Lisp, как вы, возможно, слышали).

Исследование деревьев синтаксического разбора с помощью Pytree

Но погодите – есть более удобный способ исследования структур деревьев синтаксического разбора. На рис. 18.1 показано дерево синтаксического разбора, созданное для строки «1 + 3 * (2 * 3 + 4)», изображенное в PyTree, GUI визуализации дерева,

представленном в конце предыдущей главы. Это возможно только потому, что модуль `parser2` явно строит дерево синтаксического разбора (`parser1` производит вычисления во время разбора), и благодаря общности и возможности повторного использования кода PyTree.

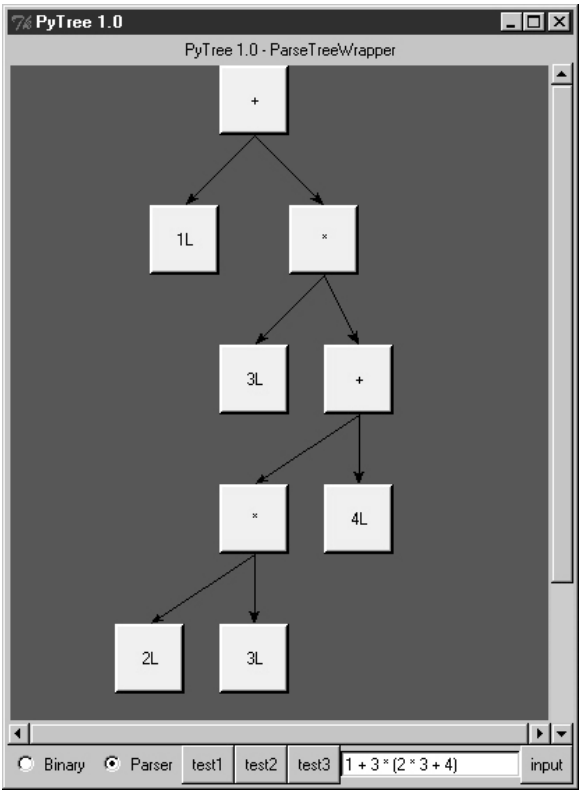


*Рис. 18.1. Дерево синтаксического разбора для 1 + 3 * (2 * 3 + 4)*

Если вы читали предыдущую главу, то вспомните, что PyTree может начертить структуру данных почти любого дерева, но настроен для работы с бинарными деревьями поиска и деревьями синтаксического разбора, которые изучаются в этой главе. Вспомните также, что при щелчке по узлу отображаемого дерева синтаксического разбора вычисляется выходящее из него поддерево. На рис. 18.2 показано всплывающее окно, генерируемое при щелчке по корневому узлу дерева (при щелчке в других частях дерева будут получены другие результаты, так как будут вычисляться меньшие поддеревья).

Рис. 18.2. Щелчок по корневому узлу, вычисляющий дерево

PyTree облегчает изучение и экспериментирование с парсером. Для определения формы дерева, получающейся для заданного выражения, запустите PyTree, щелкните по переключателю Parser, введите выражение в поле ввода внизу и нажмите «input» (или клавишу <Enter>). Для генерации дерева по введенным данным запускается класс парсера, и GUI выводит результат. Например, на рис. 18.3 показано дерево, которое генерируется, если убрать скобки из первого выражения в поле вода. Вычисление в корневом узле на этот раз дает значение 23 из-за изменения порядка вычислений, соответствующего другой форме дерева.

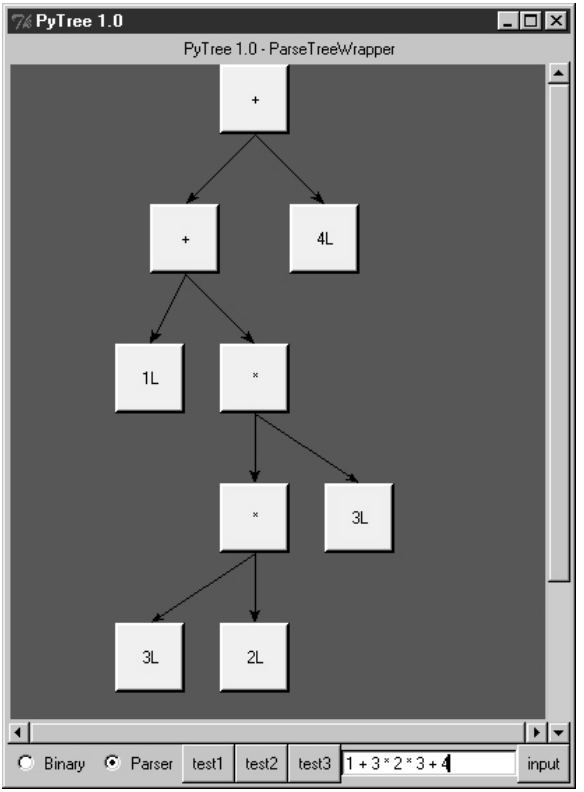


*Рис. 18.3. Дерево синтаксического разбора для 1 + 3 * 2 * 3 + 4, результат=23*

Чтобы сгенерировать дерево еще более отличающейся формы, введите в выражение дополнительные скобки и снова нажмите клавишу <Enter>. На рис. 18.4 показана еще более плоская структура дерева, получаемая при добавлении нескольких скобок, отменяющих старшинство операторов. Поскольку эти скобки изменяют форму дерева, они также снова изменяют итоговый результат выражения. Рис. 18.5 показывает окно результата, всплывающее после щелчка по корневому узлу в этом экране.

В зависимости от операторов, используемых в выражении, некоторые очень сильно различающиеся по форме деревья дают при вычислении одинаковый результат. Например, рис. 18.6 показывает дерево с перевесом в левую сторону, сгенерированное из другой строки выражения, которое, тем не менее, дает значение 56.

Наконец, на рис. 18.7 показан разобранный оператор присваивания; щелчок по корню «set» присваивает значение переменной `spam`, а щелчок по узлу `spam` выдает ре-

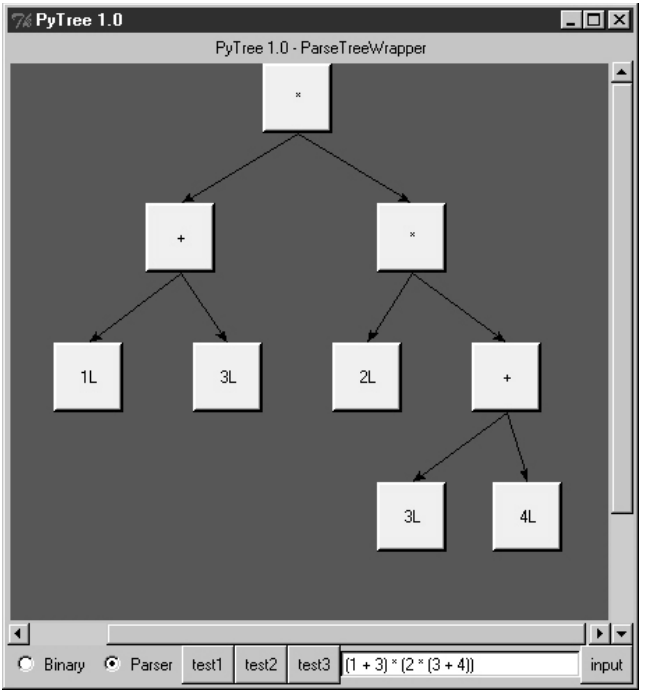


*Рис. 18.4. Дерево синтаксического разбора для «(1 + 3) * (2 * (3 + 4))»*

Рис. 18.5. Щелчок по корневому узлу и вычисление выражения

зультат −4. Если вам трудно разобраться в работе парсера, поработайте таким способом с PyTree на своем компьютере, чтобы лучше представить себе процесс синтаксического анализа. (Мне бы хотелось привести другие примеры деревьев, но к настоящему моменту я исчерпал отведенные для этого страницы.)

Парсеры в сравнении с Python

Показанные выше самостоятельные программы парсеров демонстрируют некоторые интересные понятия и подчеркивают мощь Python в универсальном программировании. Они могут оказаться типичными для тех задач, которые вы решаете полностью с помощью обычных языков типа C. Парсеры являются важным компонентом в широком круге приложений, но в некоторых случаях они не столь необходимы, как может показаться. Позвольте объяснить почему.

К данному времени, начав с анализатора выражений, мы добавили интерпретатор дерева синтаксического разбора, чтобы легче было модифицировать код. В настоящем

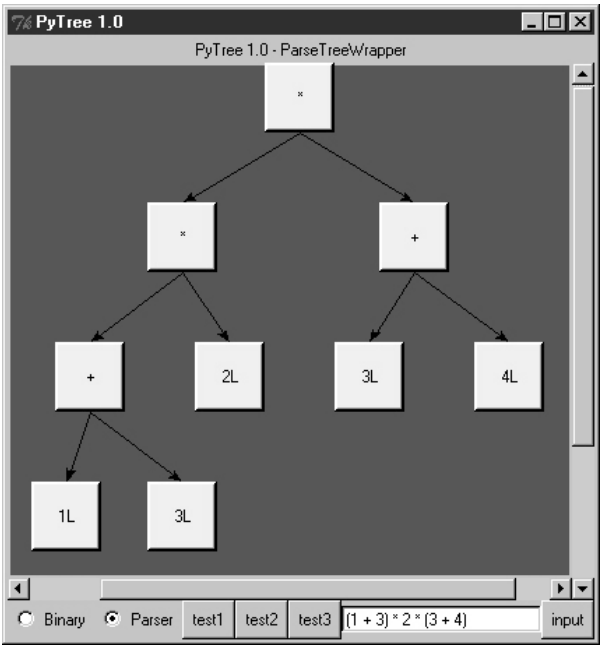


*Рис. 18.6. Дерево синтаксического разбора для «(1 + 3) * 2 * (3 + 4)», результат=56*

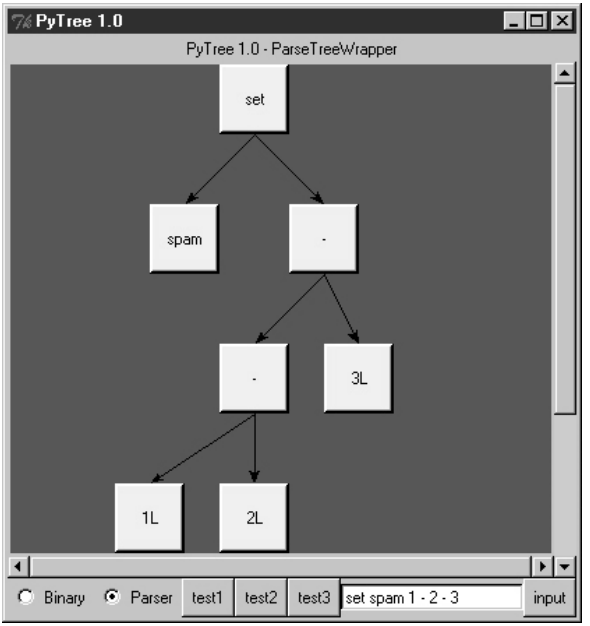


Рис. 18.7. Присваивание, левая группировка: «set spam 1 – 2 – 3»

виде парсер работает, но может быть медленнее в сравнении с реализацией на C. Если парсер используется часто, можно ускорить его работу, частично переместив в моду-

ли расширений C. Например, сначала можно перевести на C сканер, потому что он часто вызывается из парсера. В конце концов, можно добавить в грамматику компоненты, позволяющие выражениям обращаться к специфическим для приложения переменным и функциям.

Все это – путь к хорошему проектированию. Но для некоторых приложений такой подход может быть не самым лучшим в Python. Простейшим способом вычисления вводимого выражения в Python часто оказывается вызов встроенной функции `eval`. На самом деле обычно можно заменить всю программу вычисления выражения одним вызовом функции. В следующем примере демонстрируется, как это сделать.

Важна центральная идея, лежащая в основе языка и подчеркиваемая в следующем разделе: если у вас уже есть расширяемая, встраиваемая система языка высокого уровня, зачем изобретать еще одну? Python сам часто в состоянии удовлетворить потребности основанных на языке компонентов.

PyCalc: программа/объект калькулятора

В завершение этой главы я хочу показать практическое применение технологии синтаксического анализа, с которой мы познакомились в предыдущем разделе. Этот раздел представляет *PyCalc* – программу-калькулятор на Python с графическим интерфейсом, аналогичным программам калькуляторов, имеющимся в большинстве оконных систем. Но как и большинство примеров GUI в этой книге, PyCalc имеет ряд преимуществ перед существующими калькуляторами. Так как PyCalc написан на Python, его легко настраивать и переносить между оконными платформами. А так как он реализован с помощью классов, то является самостоятельной программой и библиотекой повторно используемых объектов.

GUI простого калькулятора

Однако прежде чем показывать, как написать полноценный калькулятор, начнем с более простого модуля примера 18.13. В нем реализован GUI калькулятора с ограниченными возможностями, кнопки которого просто добавляют текст в поле ввода наверху, образуя строку выражения Python. При получении и запуске строки немедленно получается результат. Рис. 18.8 показывает окно, создаваемое этим модулем при выполнении в качестве сценария верхнего уровня.

Рис. 18.8. Работа сценария calc0 под Windows (результат=160.283)

Пример 18.13. PP2E\Lang\Calculator\calc0.py

```
#!/usr/local/bin/python
# GUI простого калькулятора: немедленное выполнение выражений с помощью eval/exec

from Tkinter  import *
```

```
from PP2E.Dbase.TableBrowser.guitools import frame, button, entry

class CalcGui(Frame):
    def __init__(self, parent=None):                # расширенный фрейм
        Frame.__init__(self, parent)                # на верхнем уровне по умолчанию
        self.pack(expand=YES, fill=BOTH)            # все части расширяемы
        self.master.title('Python Calculator 0.1')  # 6 фреймов плюс поле ввода
        self.master.iconname("pcalc1")

        self.names = {}                             # пространство имен переменных
        text = StringVar()
        entry(self, TOP, text)

        rows = ["abcd", "0123", "4567", "89()"]
        for row in rows:
            frm = frame(self, TOP)
            for char in row: button(frm, LEFT, char, lambda x=text, y=char: x.set(x.get() + y))

        frm = frame(self, TOP)
        for char in "+-*/=": button(frm, LEFT, char,
                                   lambda x=text, y=char: x.set(x.get()+' '+y+' '))

        frm = frame(self, BOTTOM)
        button(frm, LEFT, 'eval',  lambda x=self, y=text: x.eval(y) )
        button(frm, LEFT, 'clear', lambda x=text: x.set('') )

    def eval(self, text):
        try:
            text.set('eval(text.get(), self.names, self.names)')
        except SyntaxError:
            try:
                exec(text.get(), self.names, self.names)
            except:
                text.set("ERROR")                   # плохо в качестве утверждения тоже?
            else:
                text.set('')                        # действие как утверждения
        except:
            text.set("ERROR")                       # другие ошибки выражения eval

if __name__== '__main__': CalcGui().mainloop()
```

Построение GUI

Вряд ли может быть калькулятор проще, но он демонстрирует основы. В этом окне есть кнопки для ввода чисел, имен переменных и операторов. Оно строится путем вставки кнопок во фреймы: каждый ряд кнопок представляет собой вложенный `Frame`, а сам GUI является подклассом `Frame`, в который вставлены `Entry` и шесть вложенных фреймов (можно было бы использовать и сетку). Фрейм калькулятора, поле ввода и кнопки сделаны расширяемыми в импортированном вспомогательном модуле `guitools`.

Этот калькулятор строит строку, которая целиком должна быть передана интерпретатору Python при нажатии кнопки «eval». Поскольку в поле ввода можно напечатать любое выражение или утверждение Python, кнопки существуют только для удобства. На самом деле поле ввода не слишком отличается от командной строки. Попробуйте ввести `import sys`, а затем `dir(sys)`, чтобы показать атрибуты модуля `sys` вверху в поле ввода – обычно калькулятор так не используют, однако это показательно.[1]

[1] И еще раз я должен предостеречь вас относительно выполнения таких строк, если вы не можете быть уверены в том, что они не нанесут вреда. Подробности смотрите в описании модуля режима ограниченного выполнения `rexec` в главе 15 «Более сложные темы Интернета».

В конструкторе `CalcGui` кнопки кодируются как списки строк; каждая строка представляет ряд, а каждый символ строки представляет кнопку. Данные обратного вызова для каждой кнопки устанавливаются как лямбда-функции со значениями аргументов по умолчанию. Функции обратного вызова сохраняют символ кнопки и связанную переменную текстового ввода, чтобы при нажатии символ мог быть добавлен в конец текущей строки графического элемента ввода.

Урок 4: встраивание лучше, чем парсеры

Вместо анализа и вычисления выражений вручную калькулятор использует `eval` и `exec` для вызова парсера/интерпретатора Python во время исполнения. По существу, калькулятор выполняет встроенный код Python из программы Python. Это возможно, потому что среда разработки Python (парсер и компилятор байт-кодов) всегда является частью систем, использующих Python. Поскольку различия между средой разработки и средой поставки нет, парсер Python может использоваться программами Python.

В результате все вычисление выражения заменяется одним вызовом `eval`. В более широком смысле, это мощный прием, о котором следует помнить: сам язык Python может заменить много небольших специальных языков. Помимо того что сокращается время разработки, клиентам приходится учить лишь один язык, который может быть достаточно прост, чтобы кодировать мог конечный пользователь.

Кроме того, Python может приобретать особенности любого приложения. Если интерфейс языка требует специфических для приложения расширений, просто добавьте классы Python или экспортируйте API для использования во встроенном коде Python в качестве расширения C. В результате вычисления кода Python со специфическими для приложения расширениями необходимость в индивидуальных парсерах почти полностью отпадает.

Есть также важное дополнительное преимущество такого подхода: у встроенного кода Python есть доступ ко всем средствам и возможностям мощного развитого языка программирования. Он может пользоваться списками, функциями, классами, внешними модулями и даже такими крупными инструментами Python, как Tkinter, полки, потоки и сокеты. Вам могут понадобиться годы, чтобы обеспечить такие функции в специальном парсере языка. Можете поинтересоваться у Гвидо.

Выполнение строк кода

Этот модуль реализует калькулятор с GUI в 45 строках кода (считая комментарии и пустые строки). Но если честно, то он мошенничает: вычисление выражений делегируется Python. На самом деле большую часть работы здесь делают встроенные функции `eval` и `exec`:

- `eval` анализирует, вычисляет и возвращает результат выражения Python, представленного в виде строки.
- `exec` выполняет произвольную команду Python, представленную в виде строки; значения не возвращается, потому что код является строкой.

Оба принимают дополнительные словари, которые могут использоваться как глобальные и локальные пространства имен для присваивания и вычисления имен, используемых в строках кода. В калькуляторе `self.names` становится таблицей символов для выполнения выражений калькулятора. Родственная функция Python `compile` может

использоваться для предварительного компилирования строк кода до передачи их в eval и exec (используйте ее, если нужно выполнять одну и ту же строку многократно).

По умолчанию пространством имен строки кода является пространство имен вызывающего. Если не передавать здесь словари, то строки выполняются в пространстве имен метода eval. Поскольку локальное пространство имен метода исчезает после возврата из вызова метода, не было бы возможности сохранить имена, присваиваемые в строке. Обратите внимание на использование встроенных обработчиков исключительных ситуаций в методе eval:

- Сначала предполагается, что строка является выражением, и выполняется встроенная функция eval.
- В случае неудачи из-за синтаксической ошибки строка вычисляется как команда с помощью exec.
- Если и это оказывается безуспешным, сообщается об ошибке в строке (синтакисическая ошибка, неопределенное имя и т. п.).

Операторы и неверные выражения могут разбираться дважды, но издержки при этом не важны, и нельзя узнать, является ли строка выражением или оператором, не разобрав ее вручную. Обратите внимание, что кнопка «eval» вычисляет выражения, но = устанавливает переменные Python, выполняя оператор присваивания. Имена переменных являются комбинациями клавиш букв abcd (или любым именем, введенным непосредственно). Они присваиваются и вычисляются в словаре, используемом для представления пространства имен калькулятора.

Расширение и прикрепление

Клиенты, повторно использующие этот калькулятор, такие же простые, как сам калькулятор. Как и большинство GUI Tkinter, основанных на классах, данный может быть расширен в подклассах – пример 18.14 приспосабливает конструктор простого калькулятора для вставки новых графических элементов.

Пример 18.14. PP2E\Lang\Calculator\calc0ext.py

```
from Tkinter import *
from calc0 import CalcGui

class Inner(CalcGui):                                  # расширение gui
   def __init__(self):
      CalcGui.__init__(self)
      Label(self,  text='Calc Subclass').pack()        # добавка after
      Button(self, text='Quit', command=self.quit).pack() # подразумевается верх

Inner().mainloop()
```

Возможно встраивание его в класс контейнера – в примере 18.15 к общему родителю прикрепляются пакет графических элементов простого калькулятора и другие.

Пример 18.15. PP2E\Lang\Calculator\calc0emb.py

```
from Tkinter  import *
from calc0 import CalcGui                          # добавить родителя, без вызовов владельца

class Outer:
    def __init__(self, parent):                     # встроить gui
        Label(parent, text='Calc Attachment').pack()  # side=top
        CalcGui(parent)                             # добавить фрейм калькулятора
        Button(parent, text='Quit', command=parent.quit).pack()
```

```
root = Tk()
Outer(root)
root.mainloop()
```

На рис. 18.9 показан результат выполнения обоих этих сценариев из разных командных строк. У обоих вверху отдельное поле ввода. Это работает, но чтобы увидеть более практическое применение такой техники повторного использования, нужно также сделать более практическим лежащий в основе калькулятор.

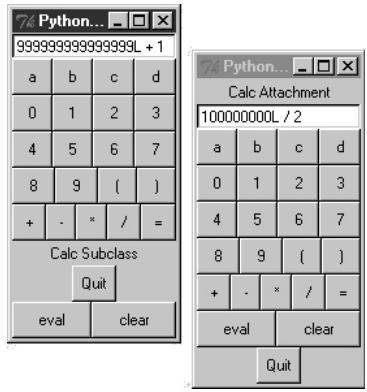


Рис. 18.9. Объект сценария calc0 прикреплен и расширен

Урок 5: повторное использование – это сила

Несмотря на простоту, графический калькулятор со встраиванием и созданием подклассов, показанный на рис. 18.9, иллюстрирует мощь Python как средства написания повторно используемого программного обеспечения. Благодаря кодированию программ с применением модулей и классов отдельно написанные компоненты почти автоматически становятся универсальными инструментами. Особенности организации программ Python способствуют написанию повторно используемого кода.

В действительности повторное использование кода является одним из главных достоинств Python и было одной из главных тем на всем протяжении этой книги. Для хорошего объектно-ориентированного проектирования нужны некоторый опыт и предусмотрительность, а выгоды, получаемые от повторного использования кода, могут не сразу стать очевидными. А иногда нам нужно быстро изменить код, а не думать о его использовании в будущем.

Но если писать код, подумывая иногда о его повторном использовании, то в долгосрочной перспективе можно сократить время разработки. Например, парсеры собственной разработки совместно использовали сканер, GUI калькулятора использует модуль `guitools`, обсуждавшийся ранее, а в следующем примере будет повторно использован класс `GuiMixin`. Иногда можно завершить часть работы, не начиная ее.

Pycalc – GUI реального калькулятора

Конечно, настоящие калькуляторы не строят строки выражений и не вычисляют их целиком; такой подход немного дает в сравнении с прославленной командной строкой

Python. Обычно выражения вычисляются по частям по мере ввода, и временные результаты отображаются сразу, как будут вычислены. Реализовать такое поведение немного труднее: выражения должны вычисляться вручную вместо однократного вызова функции `eval`. Но конечный результат становится значительно полезнее и интуитивнее.

В этом разделе представлена реализация PyCalc – программы Python/Tkinter, реализующей GUI такого традиционного калькулятора. Хотя логика его вычислений сложнее, чем у простого вышеприведенного калькулятора, он демонстрирует более развитую технологию программирования и служит интересным финалом этой главы.

Работа с PyCalc

Как обычно, сначала посмотрим GUI, а потом уже код. Запустить PyCalc можно из панелей запуска PyGadgets и PyDemos вверху дерева примеров либо непосредственно как файл *calculator.py*, листинг которого приведен ниже (например, щелкнуть по нему в менеджере файлов). На рис. 18.10 показано главное окно PyCalc. По умолчанию кнопки операндов показываются черным по голубому (а кнопки операторов – наоборот), но параметры шрифта и цвета можно передать в метод конструктора GUI. Конечно, в книге это оказывается серым по серому, поэтому нужно запустить PyCalc самостоятельно, чтобы понять, что я имею в виду.

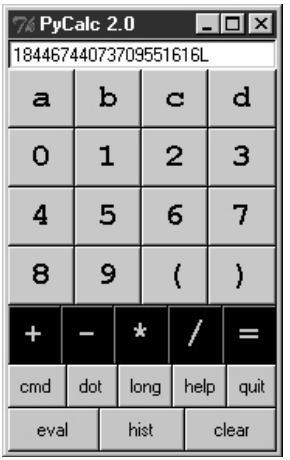


Рис. 18.10. Работа калькулятора PyCalc под Windows

Если запустить его, то можно заметить, что PyCalc реализует нормальную модель калькулятора – выражения вычисляются во время ввода, а не целиком в конце. То есть части выражений вычисляются и показываются, как только это становится возможным с учетом старшинства операторов и расставленных вручную скобок. Как действует это вычисление, я объясню через минуту.

Класс PyCalc `CalcGui` строит интерфейс GUI в виде фреймов кнопок, подобно простому калькулятору из предыдущего раздела, но в PyCalc добавляется множество новых функций. Среди них еще один ряд командных кнопок, унаследованные от `GuiMixin` методы (представленные в главе 9 «Примеры крупных GUI»), новая кнопка «cmd», выводящая немодальные диалоговые окна для ввода произвольного кода Python и всплывающее окно предыдущих вычислений. На рис. 18.11 показаны некоторые всплывающие окна PyCalc.

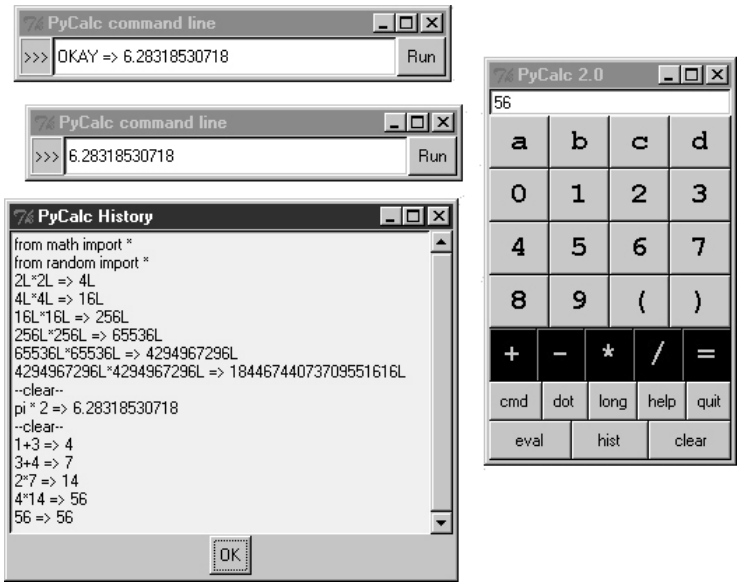


Рис. 18.11. Калькулятор PyCalc с некоторыми всплывающими окнами

Вводить выражения в PyCalc можно щелчками по кнопкам в GUI, вводя выражения целиком во всплывающих окнах командной строки или нажатием клавиш клавиатуры. PyCalc перехватывает события нажатий клавиш и интерпретирует их так же, как нажатия соответствующих кнопок; ввод <+> аналогичен нажатию кнопки <+>, клавиша пробела соответствует «clear», <Enter> соответствует «eval», забой (backspace) удаляет символ, а <?> аналогичен нажатию «help».

Всплывающие окна командной строки являются немодальными (можно вывести их в любом количестве). Они принимают любой код Python – для вычисления текста в поле ввода нажимается кнопка Run или клавиша Enter. Результат вычисления этого кода в словаре пространства имен калькулятора помещается в главное окно и может использоваться в более крупных выражениях. Можно пользоваться этим как механизмом обхода для применения внешних инструментов в расчетах. Например, в этих всплывающих окнах можно импортировать и использовать функции, написанные на Python или C. Текущее значение в главном окне калькулятора записывается также во вновь открываемых всплывающих окнах командной строки для использования во вводимых выражениях.

PyCalc поддерживает длинные целые (неограниченной точности), отрицательные и числа с плавающей точкой, поскольку это делает Python: отдельные операнды и выражения по-прежнему вычисляются встроенным вызовом `eval`, который вызывает анализатор/интерпретатор Python на этапе исполнения. Можно присваивать значения переменным и обращаться к ним в главном окне с помощью буквы, = и «eval»; присвоение выполняется в пространстве имен калькулятора (более сложные имена переменных можно вводить во всплывающих окнах командной строки). Посмотрите на использование `pi` в окне буфера: PyCalc заранее импортирует имена из модулей `math` и `random` в пространство имен, в котором вычисляются выражения.

Вычисление выражений с помощью стеков

Теперь, когда у вас есть общее представление о том, чем занимается PyCalc, я должен немного рассказать о том, как он это делает. Большинство изменений в этой версии касается организации отображения выражений и вычисления выражений. PyCalc организован в виде двух классов:

- Класс `CalcGui` управляет собственно GUI. Он контролирует события ввода и отвечает за поле отображения вверху главного окна. Однако он не вычисляет выражения; для этого он пересылает операторы и операнды, введенные в GUI, встроенному экземпляру класса `Evaluator`.
- Класс `Evaluator` управляет двумя стеками. В одном стеке записываются текущие *операторы* (например, `+`), а в другом – текущие *операнды* (например, `3.141`). Временные результаты вычисляются по мере отправки новых операторов из `CalcGui` и помещаются на стек операндов.

Из этого можно сделать вывод, что механизм вычисления выражений сводится к жонглированию стеками операторов и операндов. Во время сканирования строк выражений слева направо во время ввода операнды попутно выталкиваются на стек, но операторы служат разделителями операндов и могут порождать временные результаты перед выталкиванием. Вот общий сценарий:

- Когда обнаруживается новый оператор (то есть когда нажимается кнопка или клавиша оператора), предыдущий операнд в поле ввода выталкивается на стек операндов.
- Затем оператор добавляется в стек операторов, но только после того, как все незавершенные операторы с более высоким старшинством вытолкнуты со стека и применены к висящим операндам (например, нажатие + вызывает срабатывание всех незавершенных операторов * в стеке).
- При нажатии «eval» все оставшиеся операторы выталкиваются со стека и применяются к оставшимся операндам, и результатом является последнее оставшееся значение в стеке операндов.

В завершение последнее значение в стеке операндов показывается в поле ввода калькулятора в готовности для использования в другой операции. Этот алгоритм вычислений лучше всего, вероятно, описать на примерах. Введем пошагово несколько выражений и посмотрим, как заполняются стеки вычислений.

Трассировка стека PyCalc включается с помощью флага `debugme` в модуле; если он имеет значение «истина», стеки операторов и операндов показываются в `stdout` каждый раз, когда класс `Evaluator` собирается применить оператор и уменьшить (pop) стеки. При каждом уменьшении стека выводится кортеж, содержащий списки стеков (*operators, operands*); вершины стеков находятся в концах списков. Вот, например, вывод на консоли после ввода и вычисления простой строки:

```
1) Entered keys: "5 * 3 + 4 <eval>" [result = 19]

(['*'], ['5', '3'])    [on '+' press: displays "15"]
(['+'], ['15', '4'])   [on 'eval' press: displays "19"]
```

Обратите внимание, что незавершенное (находящееся в стеке) подвыражение `*` вычисляется при нажатии <+>: операторы `*` связывают сильнее, чем +, поэтому код вычисляется сразу перед проталкиванием оператора +. При нажатии кнопки <+> поле ввода содержит 3. Вообще говоря, поле ввода всегда содержит предыдущий операнд, когда нажимается кнопка оператора. Так как значение текстового поля проталкивается на стек операндов перед применением оператора, мы должны вытолкнуть ре-

зультаты, прежде чем выводить их после нажатия «eval» или) (в противном случае результаты проталкиваются на стек дважды):

```
2) "5 + 3 * 4 <eval>" [result = 17]

(['+', '*'], ['5', '3', '4'])   [on 'eval' press]
(['+'], ['5', '12'])            [displays "17"]
```

Здесь незавершенный + не вычисляется при нажатии кнопки *: поскольку * связывает сильнее, необходимо отложить +, пока не будет вычислен *. Оператор * выбирается со стека, когда станет виден его правый операнд. При нажатии «eval» есть два оператора, которые нужно взять со стека и применить к элементам стека операндов:

```
3) "5 + 3 + 4 <eval>" [result = 12]

(['+'], ['5', '3'])     [on the second '+']
(['+'], ['8', '4'])     [on 'eval']
```

Для таких строк операторов с одинаковым старшинством выталкивание и вычисление выполняется сразу при сканировании слева направо без откладывания вычисления. Это приводит к вычислению, ассоциативному слева, если отсутствуют скобки: 5+3+4 вычисляется как ((5+3)+4). Для операций + и * порядок не имеет значения:

```
4) "1 + 3 * ( 1 + 3 * 4 ) <eval>" [result = 40]

(['+', '*', '(', '+', '*'], ['1', '3', '1', '3', '4'])    [on ')']
(['+', '*', '(', '+'], ['1', '3', '1', '12'])             [displays "13"]
(['+', '*'], ['1', '3', '13'])                            [on 'eval']
(['+'], ['1', '39'])
```

В данном случае все операторы и операнды помещаются на стек (откладываются), пока не будет нажата кнопка) в конце. При нажатии кнопки) выталкивается со стека и вычисляется заключенное в круглые скобки подвыражение, и в поле ввода выводится 13. При нажатии «eval» вычисляется оставшаяся часть, и показывается окончательный результат (40). Результат служит левым операндом другого оператора. В действительности можно снова использовать любой временный результат: если снова нажать на кнопку оператора, не вводя новых операндов, он применяется к результату предыдущего нажатия. На рис. 18.12 показано, как выглядят оба стека в самой верхней точке во время сканирования выражения при трассировке предыдущего примера. Верхний оператор применяется к двум верхним операндам, и результат проталкивается обратно для следующего оператора:

```
5) "1 + 3 * ( 1 + 3 * 4 <eval>" [result = *ERROR*]

(['+', '*', '(', '+', '*'], ['1', '3', '1', '3', '4'])      [on eval]
(['+', '*', '(', '+'], ['1', '3', '1', '12'])
(['+', '*', '('], ['1', '3', '13'])
(['+', '*'], ['1', '*ERROR*'])
(['+'], ['*ERROR*'])
(['+'], ['*ERROR*', '*ERROR*'])
```

Эта строка возбуждает ошибку. PyCalc обрабатывает ошибки небрежно. Многие ошибки оказываются невозможными благодаря самому алгоритму, но на таких вещах, как непарные скобки, вычислитель все же спотыкается. Вместо того чтобы пытаться явно обнаруживать все возможные случаи ошибок, с помощью общего оператора try в методе reduce они перехватываются все: ошибки выражений, неопределенных имен, синтаксиса и т. д.

Операнды и временные результаты всегда помещаются на стек в виде строк, а все операторы применяются путем вызова eval. Если ошибка происходит внутри выраже-

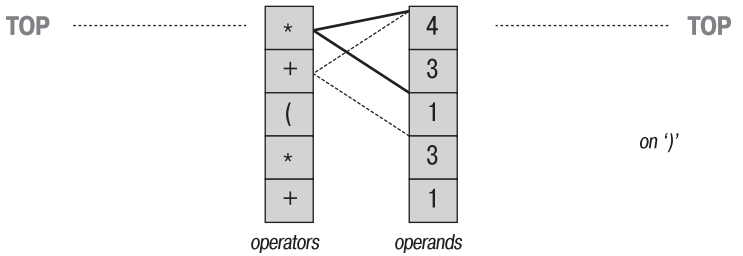


*Рис. 18.12. Стеки вычисления: 1 + 3 * (1 + 3 * 4)*

ния, проталкивается операнд *ERROR*, в результате чего все оставшиеся операторы также не могут выполниться в eval. *ERROR* распространяется до вершины выражения. В конце это оказывается последним операндом, который выводится в поле ввода, извещая о допущенной ошибке.

Исходный код PyCalc

Пример 18.16 содержит модуль исходного кода PyCalc, в котором эти идеи применены на практике в контексте GUI. Эта реализация состоит из одного файла (не считая импортированные и повторно использованные вспомогательные средства). Дополнительные детали выясняются при изучении исходного кода; и как обычно, ничто не заменит самостоятельной работы с программой для получения лучшего представления о ее функциях.

Пример 18.16. PP2E\Lang\Calculator\calculator.py

```
#!/usr/local/bin/python
##############################################################################################
# PyCalc 2.0: программа калькулятора Python/Tkinter и компонент GUI. Вычисляет выражения
# по мере ввода, перехватывает клавиши для ввода выражений; добавляет всплывающие окна
# командной строки, всплывающее окно буфера предыдущих вычислений, настройку шрифтов
# и цветов, окна подсказки и "о программе", предварительный импорт констант math/random и др.;
##############################################################################################

from Tkinter  import *                                  # графические элементы, consts
from PP2E.Gui.Tools.guimixin import GuiMixin            # метод quit
from PP2E.Dbase.TableBrowser.guitools import *          # построители графических элементов
Fg, Bg, Font = 'black', 'skyblue', ('courier', 16, 'bold')  # конфигурация по умолчанию
debugme = 1
def trace(*args):
    if debugme: print args

#############################################################################
# Основной класс - работает с интерфейсом пользователя; расширенный Frame на новом
# Toplevel или встроенный в другой элемент - контейнер
#############################################################################

class CalcGui(GuiMixin, Frame):
    Operators = "+-*/="                                 # списки кнопок
    Operands  = ["abcd", "0123", "4567", "89()"]        # настраивается

    def __init__(self, parent=None, fg=Fg, bg=Bg, font=Font):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)                # все части расширяемы
        self.eval = Evaluator()                         # встроить обработчик стека
        self.text = StringVar()                         # создать связанную переменную
```

```
        self.text.set("0")
        self.erase = 1                                  # убрать текст "0"
        self.makeWidgets(fg, bg, font)                  # построение самого gui
        if not parent or not isinstance(parent, Frame):
            self.master.title('PyCalc 2.0')             # заголовок iff владеет окном
            self.master.iconname("PyCalc")              # то же для привязок ключей
            self.master.bind('<KeyPress>', self.onKeyboard)
            self.entry.config(state='disabled')
        else:
            self.entry.config(state='normal')
            self.entry.focus()

    def makeWidgets(self, fg, bg, font):               # 7 фреймов плюс текстовое поле
        self.entry = entry(self, TOP, self.text)       # шрифт, цвет настраиваются
        for row in self.Operands:
            frm = frame(self, TOP)
            for char in row:
                button(frm, LEFT, char,
                            lambda x=self, y=char: x.onOperand(y),
                            fg=fg, bg=bg, font=font)

        frm = frame(self, TOP)
        for char in self.Operators:
            button(frm, LEFT, char,
                        lambda x=self, y=char: x.onOperator(y),
                        fg=bg, bg=fg, font=font)

        frm = frame(self, TOP)
        button(frm, LEFT, 'cmd ', self.onMakeCmdline)
        button(frm, LEFT, 'dot ', lambda x=self: x.onOperand('.'))
        button(frm, LEFT, 'long', lambda x=self: x.text.set(x.text.get()+'L'))
        button(frm, LEFT, 'help', self.help)
        button(frm, LEFT, 'quit', self.quit)               # из guimixin

        frm = frame(self, BOTTOM)
        button(frm, LEFT, 'eval ', self.onEval)
        button(frm, LEFT, 'hist ', self.onHist)
        button(frm, LEFT, 'clear', self.onClear)

    def onClear(self):
        self.eval.clear()
        self.text.set('0')
        self.erase = 1

    def onEval(self):
        self.eval.shiftOpnd(self.text.get())            # последний или единственный операнд
        self.eval.closeall()                            # применить все оставшиеся операторы
        self.text.set(self.eval.popOpnd())              # нужен pop: следующий оператор?
        self.erase = 1

    def onOperand(self, char):
        if char == '(':
            self.eval.open()
            self.text.set('(')                          # очистить текст
            self.erase = 1
        elif char == ')':
            self.eval.shiftOpnd(self.text.get()) # последний или единственный вложенный операнд
            self.eval.close()                    # тоже pop: следующий оператор?
            self.text.set(self.eval.popOpnd())
            self.erase = 1
```

```
        else:
            if self.erase:
                self.text.set(char)                     # очищает последнее значение
            else:
                self.text.set(self.text.get() + char)   # или добавить к операнду
            self.erase = 0

    def onOperator(self, char):
        self.eval.shiftOpnd(self.text.get())    # push операнд слева
        self.eval.shiftOptr(char)               # eval выражения слева?
        self.text.set(self.eval.topOpnd())      # push оператор, показать операнд|результат
        self.erase = 1                          # стереть при следующем операнде |'('

    def onMakeCmdline(self):
        new = Toplevel()                            # новое окно верхнего уровня
        new.title('PyCalc command line')            # произвольный код python
        frm = frame(new, TOP)                       # расширяется только поле Entry
        label(frm, LEFT, '>>>').pack(expand=NO)
        var = StringVar()
        ent = entry(frm, LEFT, var, width=40)
        onButton = (lambda s=self, v=var, e=ent: s.onCmdline(v,e))
        onReturn = (lambda event, s=self, v=var, e=ent: s.onCmdline(v,e))
        button(frm, RIGHT, 'Run', onButton).pack(expand=NO)
        ent.bind('<Return>', onReturn)
        var.set(self.text.get())

    def onCmdline(self, var, ent):              # eval данные окна командной строки
        try:
            value = self.eval.runstring(var.get())
            var.set('OKAY')
            if value != None:                   # выполнять в словаре пространства имен eval
                self.text.set(value)            # выражение или утверждение
                self.erase = 1
                var.set('OKAY => '+ value)
        except:                                 # результат в поле калькулятора
            var.set('ERROR')                    # состояние в поле всплывающего окна
        ent.icursor(END)                        # точка вставки после текста
        ent.select_range(0, END)                # выделить msg, чтобы следующая клавиша удалила

    def onKeyboard(self, event):
        pressed = event.char                    # при событии нажатия клавиатуры
        if pressed != '':                       # как если бы нажата кнопка
            if pressed in self.Operators:
                self.onOperator(pressed)
            else:
                for row in self.Operands:
                    if pressed in row:
                        self.onOperand(pressed)
                        break
                else:
                    if pressed == '.':
                        self.onOperand(pressed)             # может быть началом операнда
                    if pressed in 'Ll':
                        self.text.set(self.text.get()+'L')  # не может: не удалять
                    elif pressed == '\r':
                        self.onEval()                       # клавиша enter = eval
                    elif pressed == ' ':
                        self.onClear()                      # пробел = clear
                    elif pressed == '\b':
```

```
                    self.text.set(self.text.get()[:-1])  # backspace
                elif pressed == '?':
                    self.help()

    def onHist(self):
        # show recent calcs log popup
        # self.infobox('PyCalc History', self.eval.getHist())
        from ScrolledText import ScrolledText
        new = Toplevel()                                # создать новое окно
        ok = Button(new, text="OK", command=new.destroy)
        ok.pack(pady=1, side=BOTTOM)                    # пакуется первым=обрезается последним
        text = ScrolledText(new, bg='beige')            # добавить Text + полоса прокрутки
        text.insert('0.0', self.eval.getHist())         # получить текст Evaluator
        text.pack(expand=YES, fill=BOTH)

        # new window goes away on ok press or enter key
        new.title("PyCalc History")
        new.bind("<Return>", (lambda event, new=new: new.destroy()))
        ok.focus_set()                      # сделать новое окно модальным:
        new.grab_set()                      # получить фокус клавиатуры, захватить приложение
        new.wait_window()                   # не возвращаться до new.destroy

    def help(self):
        self.infobox('PyCalc',  'PyCalc 2.0\n'
                                'A Python/Tk calculator\n'
                                'August, 1999\n'
                                'Programming Python 2E\n\n'
                                'Use mouse or keyboard to\n'
                                'input numbers and operators,\n'
                                'or type code in cmd popup')

##############################################################
# Класс вычислителя выражений встроен в экземпляр CalcGui
# и используется для выполнения вычислений
##############################################################

class Evaluator:
    def __init__(self):
        self.names = {}                          # пространство имен для своих переменных
        self.opnd, self.optr = [], []            # два пустых стека
        self.hist = []                           # журнал предыдущих вычислений
        self.runstring("from math import *")     # заранее импортировать модули математики
        self.runstring("from random import *")   # в пространство имен калькулятора

    def clear(self):
        self.opnd, self.optr = [], []            # оставить имена целыми names intact
        if len(self.hist) > 64:                  # ограничить размер буфера
            self.hist = ['clear']
        else:
            self.hist.append('--clear--')

    def popOpnd(self):
        value = self.opnd[-1]                   # pop/возврат верхнего|последнего операнда
        self.opnd[-1:] = []                     # вывести и сдвинуть
        return value

    def topOpnd(self):
        return self.opnd[-1]                    # верхний операнд (конец списка)

    def open(self):
        self.optr.append('(')                   # обращаться с '(' как с оператором
```

```
    def close(self):                            # по `)` вытолкнуть до самой верхней `(`
        self.shiftOptr(`)`)                     # ok, если пусто: остается пусто
        self.optr[-2:] = []                     # pop или снова добавляется оператором

    def closeall(self):
        while self.optr:                        # остальные принудительно по `eval`
            self.reduce()                       # последним может быть имя переменной
        try:
            self.opnd[0] = self.runstring(self.opnd[0])
        except:
            self.opnd[0] = `*ERROR*`            # pop, иначе снова добавится:

    afterMe = {`*`: [`+`, `-`, `(`, `=`],       # член класса
               `/`: [`+`, `-`, `(`, `=`],       # не выталкивать операторы для клавиши
               `+`: [`(`, `=`],                 # если это предыдущий оператор: push
               `-`: [`(`, `=`],                 # иначе: pop/eval предыдущий оператор
               `)`: [`(`, `=`],                 # все ассоциативны слева
               `=`: [`(`] }

    def shiftOpnd(self, newopnd):               # push операнд для оператра, `)`, eval
        self.opnd.append(newopnd)

    def shiftOptr(self, newoptr):               # применить операции с <= старшинством
        while (self.optr and
               self.optr[-1] not in self.afterMe[newoptr]):
            self.reduce()
        self.optr.append(newoptr)       # push оператор над результатом
                                        # операторы предполагают, что следующий операнд стирает
    def reduce(self):
        trace(self.optr, self.opnd)
        try:                                    # свернуть верхнее выражение
            operator       = self.optr[-1]      # pop верхний оператор (в конце)
            [left, right]  = self.opnd[-2:]     # pop 2 верхних операнда (в конце)
            self.optr[-1:] = []                 # удалить срез по месту
            self.opnd[-2:] = []
            result = self.runstring(left + operator + right)
            if result == None:
                result = left               # присваивание? клавиша имени переменной
            self.opnd.append(result)        # push строку результата назад
        except:
            self.opnd.append(`*ERROR*`)     # ошибка стека/числа/имени

    def runstring(self, code):
        try:
            result = `eval(code, self.names, self.names)`  # опробовать
                                                           # выражение: строка
            self.hist.append(code + ` => ` + result)       # добавить в журнал
        except:
            exec code in self.names, self.names            # опробовать утверждение: None
            self.hist.append(code)
            result = None
        return result

    def getHist(self):
        import string
        return string.join(self.hist, `\n`)

def getCalcArgs():
    from sys import argv
    config = {}                         # получить аргументы командной строки в словаре
```

```
    for arg in argv[1:]:                # пример: -bg black -fg red
        if arg in ['-bg', '-fg']:       # шрифт пока не поддерживается
            try:
                config[arg[1:]] = argv[argv.index(arg) + 1]
            except:
                pass
    return config

if __name__ == '__main__':
    apply(CalcGui, (), getCalcArgs()).mainloop()  # по умолчанию окно верхнего уровня
```

Использование PyCalc как компонента

PyCalc выполняется как самостоятельная программа на моей машине, но он полезен и в контексте других GUI. Как и большинство классов GUI в этой книге, PyCalc можно индивидуально настраивать с помощью расширений подклассов или встраивать в более крупные GUI. Модуль примера 18.17 демонстрирует один из способов повторного использования класса PyCalc `CalcGui` путем расширения и встраивания подобно тому, как это делалось выше для простого калькулятора.

Пример 18.17. PP2E\Lang\Calculator\calculator_test.py

```
###############################################################################
# Проверка использования калькулятора как расширяемого и встраиваемого компонента gui;
###############################################################################

from Tkinter import *
from calculator import CalcGui
from PP2E.Dbase.TableBrowser.guitools import *

def calcContainer(parent=None):
    frm = Frame(parent)
    frm.pack(expand=YES, fill=BOTH)
    Label(frm, text='Calc Container').pack(side=TOP)
    CalcGui(frm)
    Label(frm, text='Calc Container').pack(side=BOTTOM)
    return frm

class calcSubclass(CalcGui):
    def makeWidgets(self, fg, bg, font):
        Label(self, text='Calc Subclass').pack(side=TOP)
        Label(self, text='Calc Subclass').pack(side=BOTTOM)
        CalcGui.makeWidgets(self, fg, bg, font)
        #Label(self, text='Calc Subclass').pack(side=BOTTOM)

if __name__ == '__main__':
    import sys
    if len(sys.argv) == 1:              # % calculator_test.py
        root = Tk()                     # запуск 3 калькуляторов в одном процессе
        CalcGui(Toplevel())             # каждый в новом окне верхнего уровня
        calcContainer(Toplevel())
        calcSubclass(Toplevel())
        Button(root, text='quit', command=root.quit).pack()
        root.mainloop()
    if len(sys.argv) == 2:              # % calculator_testl.py -
        CalcGui().mainloop()            # как самостоятельное окно (root по умолчанию)
    elif len(sys.argv) == 3:            # % calculator_test.py - -
        calcContainer().mainloop()      # как встроенный компонент
    elif len(sys.argv) == 4:            # % calculator_test.py - - -
        calcSubclass().mainloop()       # как настраиваемый суперкласс
```

На рис. 18.13 показан результат выполнения этого сценария без аргументов командной строки. Получаем экземпляры первоначального класса калькулятора плюс класс контейнера и подкласс, определенные в этом сценарии, которые все прикреплены к новым окнам верхнего уровня.

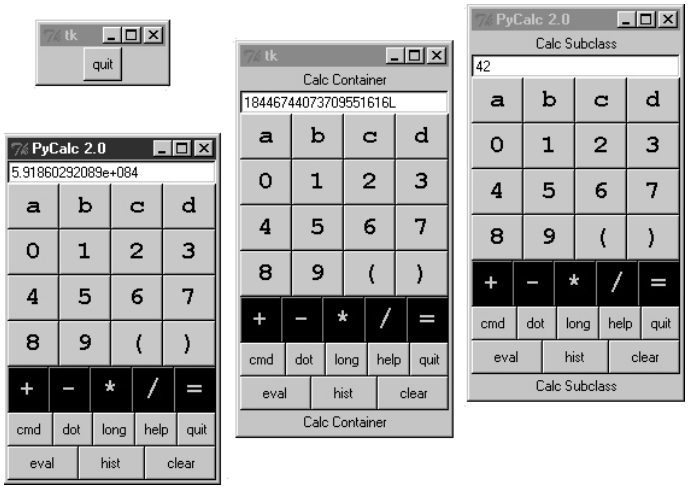


Рис. 18.13. Сценарий calculator_test: прикрепление и расширение

Два правых окна повторно используют базовый код PyCalc, выполняющийся в левом окне. Все эти окна выполняются в одном процессе (например, при завершении одного завершаются все), но все они действуют как независимые окна. Обратите внимание, что при таком выполнении трех калькуляторов в одном процессе у каждого из них собственное пространство имен для вычисления выражений, потому что это атрибут экземпляра класса, а не глобальная переменная уровня модуля. Поэтому установка переменных в одном калькуляторе касается только этого калькулятора и не изменяет значений, присваиваемых в других окнах. Аналогично у каждого калькулятора есть собственный управляющий объект стека вычислений, благодаря чему вычисления в одном окне не показываются в других окнах и не влияют на них.

Два расширения в этом сценарии, конечно, искусственные – они просто добавляют метки вверху и внизу окна – но концепция широко применима. Можно повторно использовать класс калькулятора, прикрепляя его к любому GUI, где требуется калькулятор, и произвольным образом настроить его с помощью подклассов. Это повторно используемый графический элемент.

Помещение новых кнопок в новые компоненты

Одно очевидное повторное использование калькулятора может состоять в добавлении новых функциональных кнопок выражений – квадратных корней, обратных значений, кубов и т. д. Такие операции можно вводить во всплывающих окнах командной строки, но кнопки несколько более удобны. Такие функции можно добавить и в саму основную реализацию калькулятора; но поскольку набор таких полезных функций может различаться в зависимости от пользователей и приложений, более правильным подходом явится их добавление в отдельные расширения. Например, класс примера 18.18 добавляет в PyCalc несколько новых кнопок путем встраивания его (то есть прикрепления) в контейнер.

Пример 18.18. PP2E\Lang\Calculator\calculator_plus_emb.py

```
##############################################################################################
# Контейнер с дополнительным рядом кнопок для частых операций; более практичная настройка:
# для новых операций (sqrt, 1/x и т. д.) добавляются путем встраивания/композиции, а не создания
# подклассов; новые кнопки добавляются после всего фрейма CalGui из-за порядка упаковки/опций;
##############################################################################################

from Tkinter import *
from calculator import CalcGui, getCalcArgs
from PP2E.Dbase.TableBrowser.guitools import frame, button, label

class CalcGuiPlus(Toplevel):
    def __init__(self, **args):
        Toplevel.__init__(self)
        label(self, TOP, 'PyCalc Plus - Container')
        self.calc = apply(CalcGui, (self,), args)
        frm = frame(self, BOTTOM)
        extras = [('sqrt', 'sqrt(%s)'),
                  ('x^2 ',  '(%s)**2'),
                  ('x^3 ',  '(%s)**3'),
                  ('1/x ',  '1.0/(%s)')]
        for (lab, expr) in extras:
            button(frm, LEFT, lab, (lambda m=self.onExtra, e=expr: m(e)) )
        button(frm, LEFT, ' pi ', self.onPi)
    def onExtra(self, expr):
        text = self.calc.text
        eval = self.calc.eval
        try:
            text.set(eval.runstring(expr % text.get()))
        except:
            text.set('ERROR')
    def onPi(self):
        self.calc.text.set(self.calc.eval.runstring('pi'))

if __name_== '__main__':
    root = Tk()
    button(root, TOP, 'Quit', root.quit)
    apply(CalcGuiPlus, (), getCalcArgs()).mainloop()     # -bg,-fg в calcgui
```

Так как PyCalc закодирован как класс Python, аналогичного эффекта всегда можно добиться, расширив PyCalc в новый подклас вместо его встраивания, как показано в примере 18.19.

Пример 18.19. PP2E\Lang\Calculator\calculator_plus_ext.py

```
##############################################################################################
# Настройка, добавляющая ряд кнопок частых операций; более практичная настройка:  кнопки
# для новых операций (sqrt, 1/x и т. д.) добавляются путем создания подклассов, а не встраивания;
# новые кнопки показываются перед фреймом, прикрепленным к низу класса calcgui;
##############################################################################################

from Tkinter import *
from calculator import CalcGui, getCalcArgs
from PP2E.Dbase.TableBrowser.guitools import *

class CalcGuiPlus(CalcGui):
    def makeWidgets(self, *args):
        label(self, TOP, 'PyCalc Plus - Subclass')
        apply(CalcGui.makeWidgets, (self,) + args)
        frm = frame(self, BOTTOM)
```

```
        extras = [('sqrt', 'sqrt(%s)'),
                  ('x^2 ', '(%s)**2'),
                  ('x^3 ', '(%s)**3'),
                  ('1/x ', '1.0/(%s)')]
        for (lab, expr) in extras:
            button(frm, LEFT, lab, (lambda m=self.onExtra, e=expr: m(e)) )
        button(frm, LEFT, ' pi ', self.onPi)
    def onExtra(self, expr):
        try:
            self.text.set(self.eval.runstring(expr % self.text.get()))
        except:
            self.text.set('ERROR')
    def onPi(self):
        self.text.set(self.eval.runstring('pi'))

if __name__== '__main__':
    apply(CalcGuiPlus, (), getCalcArgs()).mainloop()     # passes -bg, -fg on
```

Обратите внимание, что функции обратного вызова этих кнопок используют `1.0/x` для принудительного деления с плавающей точкой при обращении величин (деление целых отсекает остатки) и заключают значения поля ввода в скобки (чтобы обойти проблемы старшинства). Вместо этого можно было бы преобразовывать текст поля ввода в число и выполнять операции с действительными числами, но Python автоматически проделывает всю работу, когда строки выражений выполняются в необработанном виде.

Также обратите внимание, что добавляемые в этом сценарии кнопки действуют просто над текущим значением поля ввода непосредственно. Это не совсем то же самое, что операторы выражений, применяемые со стековым вычислителем (чтобы сделать их настоящими операторами, нужны дополнительные переделки). Тем не менее эти кнопки подтверждают то, что должны продемонстрировать эти сценарии – они используют PyCalc как компонент, как снаружи, так и внутри.

Наконец, для тестирования расширенных классов калькулятора и параметров конфигурации PyCalc сценарий примера 18.20 показывает четыре отдельных окна калькулятора (этот сценарий запускает PyDemos).

Пример 18.20. PP2E\Lang\Calculator\calculator_plusplus.py

```
#!/usr/local/bin/python
from Tkinter import Tk, Button, Toplevel
import calculator, calculator_plus_ext, calculator_plus_emb

# демонстрация одновременно 3 разновидностей калькулятора
# каждый является отдельным объектом калькулятора и окном

root=Tk()
calculator.CalcGui(Toplevel())
calculator.CalcGui(Toplevel(), fg='white', bg='purple')
calculator_plus_ext.CalcGuiPlus(Toplevel(), fg='gold', bg='black')
calculator_plus_emb.CalcGuiPlus(fg='black', bg='red')
Button(root, text='Quit Calcs', command=root.quit).pack()
root.mainloop()
```

Рис. 18.14 показывает результат – четыре независимых калькулятора в окнах верхнего уровня в одном и том же процессе. Окна слева и справа представляют специализированное повторное использование PyCalc как компонента. Хотя в данной книге это может быть невидно, во всех четырех используются разные цветовые схемы; классы калькулятора принимают параметры настройки цвета и шрифта и при необходимости передают их по цепочке вызовов.

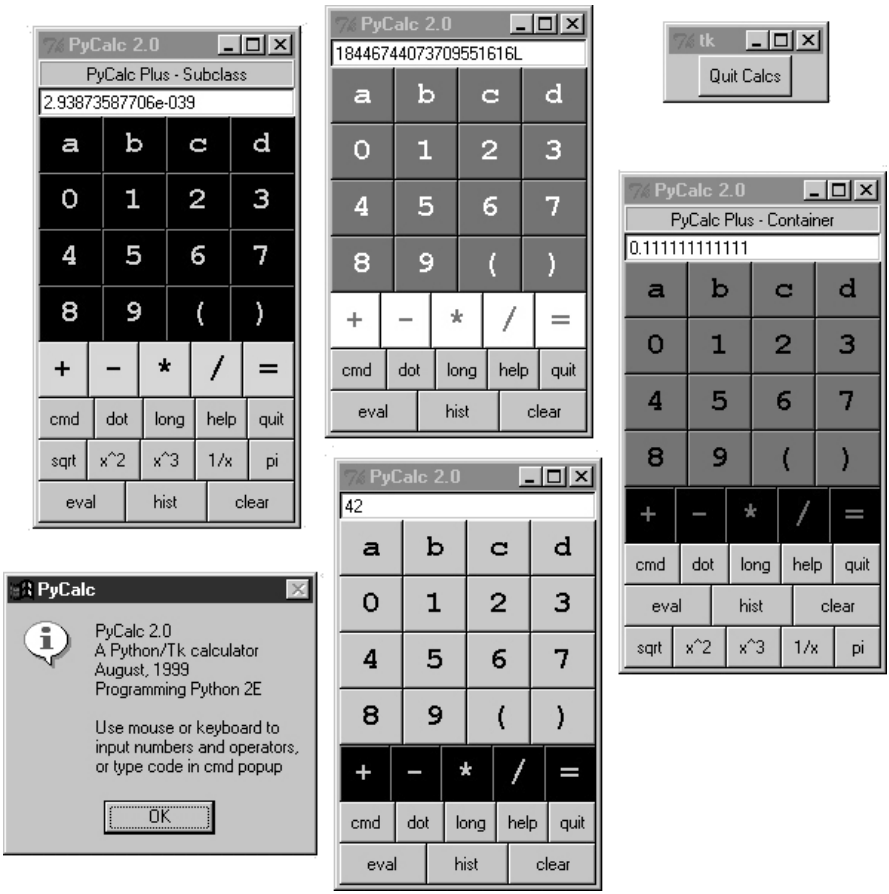


Рис. 18.14. Сценарий calculator_plusplus: расширение, встраивание и настройка

Как мы узнали ранее, эти калькуляторы могут также выполняться в виде независимых процессов при порождении командных строк с помощью модуля `launchmodes`, с которым мы познакомились в главе 3 «Системные средства параллельного выполнения». В действительности панели запуска PyGadgets и PyDemos так и запускают калькуляторы, поэтому смотрите подробности в их коде.

Урок 6: получайте удовольствие

В заключение об одном менее ощутимом, но важном аспекте программирования на Python. Пользователи-новички часто отмечают, что на Python легко «сказать, что тебе нужно», не увязнув при этом в сложном синтаксисе или неясных правилах. Это язык, дружественный программисту. Действительно, не столь уж редко программы Python работают с первой попытки.

Как мы видели в этой книге, такая отличительная особенность обусловлена рядом факторов – отсутствием объявлений и этапа компиляции, простым синтаксисом, практичными встроенными объектами и т. д. Python специально спроектирован для ускорения разработки (мысль, которую мы подробнее обсудим в главе 21 «Заключение: Python и цикл разработки»). Для многих пользователей конечным результатом оказывается весьма выразительный и отзывчивый язык, пользоваться которым удовольствие.

Например, приведенные выше программы калькуляторов были первоначально набросаны за день исходя из неясных и незаконченных целей. Не было аналитического этапа, формального проекта и формальной стадии кодирования. Я ввел некоторые идеи с клавиатуры, и они заработали. Более того, интерактивная природа Python позволила мне экспериментировать с новыми идеями и немедленно получать результат. После первоначальной разработки калькулятор был отшлифован и дополнен, но основа реализации сохранилась неизменной.

Естественно, такой спокойный режим программирования годится не для каждого проекта. Иногда оправданно уделить предварительному проектированию больше внимания. Для более требовательных задач в Python есть модульные конструкции и поддерживаются системы, которые могут расширяться на Python или C. А GUI простого калькулятора многие, возможно, не сочтут «серьезной» программной разработкой. Но такие программы тоже нужны.

V

Интеграция

В этой части книги изучаются интерфейсы Python для связи с программными компонентами, написанными на других языках. Особое внимание уделено объединению Python с программами, написанными на C и C++, но попутно представлены и другие технологии интеграции. Эта часть состоит из двух глав, соответствующих двум главным способам интеграции Python с C:

- Глава 19 «Расширяем Python». Эта глава представляет средства, позволяющие сценариям Python обращаться к компонентам C. Компоненты C принимают вид новых модулей или типов объектов. В этой главе также рассказывается о SWIG – системе, автоматически генерирующей связующий код, необходимый для экспорта библиотек C и C++ в сценарии Python, и в значительной мере скрывающей сложности, лежащие в основе расширений.
- Глава 20 «Встраиваем Python». Эта глава представляет средства, позволяющие программам C выполнять сценарии Python. Эти средства находятся в API исполнения Python – наборе функций, предоставляемых интерпретатором Python и присоединяемых к программам C/C++. Глава завершается кратким рассмотрением других тем и систем, связанных с интеграцией – Jpython, COM, CORBA и т. д.

Эта часть книги предполагает умение читать программы C и полезна в основном разработчикам, занимающимся реализацией слоев интеграции приложений, передающих управление в сценарии Python и из них. Однако поскольку компоненты C лежат в центре многих систем Python, базовое понимание концепций интеграции может оказаться полезным даже для тех, кто кодирует исключительно на Python.

19

Расширяем Python

«Я заблудился в C»

До сих пор в этой книге мы работали с тем Python, который получили в готовом виде. Мы пользовались интерфейсами к внешним сервисам и кодировали расширения в виде модулей Python. Но мы не вводили каких-либо внешних сервисов помимо встроенного набора. Для многих пользователей это совершенно обоснованно: для такого автономного программирования главным образом и применяют Python. Как мы видели, Python поставляется в полном комплекте – с интерфейсами к системным средствам, протоколами Интернета, GUI, файловыми системами и многим другим.

Но для многих систем важнейшей характеристикой языка является возможность интеграции Python с C-совместимыми компонентами. На самом деле роль Python как языка расширений и интерфейсов в больших системах является одной из причин его популярности и того, что его часто называют «управляющим» (scripting) языком. Его архитектура поддерживает *гибридные* системы, в которых смешаны компоненты, написанные на разных языках программирования. Так как у каждого языка есть свои сильные стороны, возможность выбирать на покомпонентной основе является мощной концепцией. Python можно включать во все системы, где требуется простой и гибкий языковый инструмент.

Например, компилируемые языки типа C и C++ оптимизированы по скорости *выполнения*, но программировать на них сложно и разработчикам, и, особенно, конечным пользователям. Python оптимизирован по скорости *разработки*, поэтому при использовании сценариев Python для управления или индивидуальной подгонки программных компонентов, написанных на C или C++, получаются более гибкие системы и резко ускоряется разработка. Системы, спроектированные так, что их подгонка возлагается на сценарии Python, не требуют поставки с полного исходного кода и изучения конечными пользователями сложных или специальных языков. Кроме того, переведя часть компонентов с чистого Python на C, можно оптимизировать производительность программ.

Темы, относящиеся к интеграции

Последние две технические главы в этой книге знакомят со средствами Python, позволяющими взаимодействовать с внешним миром, и рассказывают о возможности его применения в качестве встроенного языкового инструмента в других системах и интерфейсах для расширения сценариев Python новыми модулями и типами, реализуемыми на C-совместимых языках. Я также кратко расскажу о других, менее связанных с C технологиях интеграции, таких как COM и JPython.

При соединении Python с компонентами C на верхнем уровне может оказаться либо Python, либо C, поэтому есть два разных API интеграции:

- Интерфейс *расширения* для выполнения расширений C из программ Python
- Интерфейс *встраивания* для выполнения кода Python из C-программ

В данной главе рассказывается о расширении, а в следующей – о встраивании. В некоторых системах используется одна из этих схем, но во многих – обе. Например, встроенный код Python, выполняемый из C, может также использовать присоединенные расширения C для взаимодействия с содержащим его приложением. А в системах, основанных на обратных вызовах, код C, доступ к которому осуществляется через интерфейсы расширения, в дальнейшем может использовать технику встраивания для выполнения обработчиков обратного вызова Python. Архитектура Python открыта и реентерабельна, что позволяет соединять языки произвольным образом.

Прежде чем заняться деталями, хочу заметить, что тема интеграции Python и C обширна – в принципе, интерфейс выполнения для системы Python состоит из всего набора ее *экспортируемых* (*extern*) функций C. Следующие две главы сосредоточены только на средствах, обычно применяемых для интеграции с внешними модулями. Дополнительные примеры, выходящие за рамки этой книги и прилагаемого CD, можно найти в самом исходном коде Python; богатым источником кода служат его каталоги `Modules` и `Objects`. Большинство встроенных элементов Python, использованных в этой книге – от простых, типа целых чисел и строк, до более сложных, вроде файлов, системных вызовов, Tkinter и DBM, – используют интегрирующие API и могут быть изучены по дистрибутиву исходного кода Python.

В этих главах предполагается знание основных понятий программирования на C. В противном случае можно без особого ущерба пропустить или бегло просмотреть эти главы. Обычно разработчики на C пишут для системы код интерфейсов расширения и встраивания, а основную часть программирования системы с помощью только Python выполняют другие. Но если вы в достаточной мере знакомы с программированием на C, чтобы распознать необходимость в языке расширения, то, вероятно, у вас есть фундаментальные знания, необходимые для изучения этой главы. Приятным обстоятельством для обеих глав является то, что значительная часть сложностей, присущих интеграции Python со статическим компилируемым языком типа C, может быть автоматизирована с помощью таких инструментов, как SWIG, в сфере расширений и API высокого уровня в области встраивания.

Обзор расширений на C

Поскольку сам Python сегодня написан на C, компилированные расширения Python могут быть написаны на любых языках, совместимых с C в отношении стеков вызова и компоновки. В их число входят C, а также C++ с надлежащими объявлениями «extern C» (автоматически помещаемыми в файлы заголовков Python). Есть два вида расширений Python на совместимых с C языках:

- *Модули на C*, воспринимаемые своими клиентами как файлы модулей Python
- *Типы C*, ведущие себя, как встренные типы (числа, списки и т. д.)

Обычно модули на C используются для реализации простых библиотек функций, а типы C используются для программирования объектов, генерирующих множественные экземпляры. Так как встроенные типы в действительности являются просто закодированными в расширениях на C, у вновь создаваемых типов на C такие же возможности, как и у встроенных типов: вызовы методов, добавление, обращение по ин-

дексу, срезы и т. п.[1] Тем не менее, в текущей версии Python типы не совсем являются классами – нельзя дополнительно настраивать типы, создавая подклассы Python, кроме как добавляя классы-оболочки в качестве клиентских интерфейсов к ним.[2] Подробнее об этом ниже.

Как модули, так и типы C регистрируют свои операции в интерпретаторе Python в виде указателей функций C. Слой C во всех случаях отвечает за преобразование аргументов, передаваемых из Python в формат C, и преобразование результатов из формата C в формат Python. Сценарии Python просто экспортируют расширения C и пользуются им так, как если бы они в действительности были написаны на Python; все действия по трансляции осуществляет код C.

Модули и типы на C ответственны также за сообщение Python об ошибках, обнаруживая ошибки, генерируемые вызовами Python API, и управляя счетчиками ссылок сборщика мусора для объектов, сохраняемых слоем C неограниченно долго – объекты Python, хранимые в вашем коде C, не будут убраны как мусор, пока вы обеспечите сохранение их счетчиками ссылок значений, больших нуля. Модули и типы C могут компоноваться с Python статически (на этапе сборки) или динамически (при первом импортировании).

Простой модуль расширения на C

Такова история вкратце, но чтобы сделать ее более конкретной, нужно обратиться к коду. Типы C обычно экспортируют модуль C с функцией конструктора. По этой причине (и потому, что они проще) начнем с изучения основ программирования *модулей* C на коротком примере.

При добавлении новых или существующих компонентов C в Python необходимо написать на C слой интерфейсной («glue») логики, обеспечивающий межъязыковую диспетчеризацию и трансляцию данных. Файл исходного кода на C примера 19.1 показывает, как написать такой слой вручную. В нем реализован простой модуль расширения на C с именем `hello` для использования в сценариях Python, содержащий функцию с именем `message`, которая просто возвращает передаваемую ей в качестве аргумента строку вместе с добавленным в ее начало некоторым текстом.

Пример 19.1. PP2E\Integrate\Extend\Hello\hello.c

```
/*****************************************************************
 * Простой модуль расширения на C для Python с именем "hello"; скомпилируйте
 * его в ".so" в пути поиска python, импортируйте и вызовите hello.message;
 *****************************************************************/

#include <Python.h>
#include <string.h>

/* функции модуля */
static PyObject *                               /* возвращает объект */
message(PyObject *self, PyObject *args)         /* self не используется в модулях */
{                                               /* аргументы вызова python */
```

[1] Да, каждый раз, создавая в Python целое число или строку, вы генерируете новый объект экземпляра типа, описанного на C (даже если не замечаете этого). Это не столь уж неэффективно, как может показаться; как мы увидим, операции с типами диспетчеризуются через быстрые указатели C, и Python внутренне кэширует часть целых чисел и строк, чтобы по возможности избежать создания объектов.

[2] В современных версиях Python встроенные типы вполне объектно-ориентированы, и есть возможность наследовать в пользовательских классах классы встроенных типов Python.

```
  char *fromPython, result[64];
  if (! PyArg_Parse(args, "(s)", &fromPython))  /* преобразование Python -> C */
        return NULL;                            /* null=возбудить исключительную ситуацию */
  else {
        strcpy(result, "Hello, ");              /* создать строку C */
        strcat(result, fromPython);             /* добавить переданную строку Python */
        return Py_BuildValue("s", result);      /* преобразовать C -> Python */
  }
}

/* таблица регистрации */
static struct PyMethodDef hello_methods[] = {
  {"message", message, 1},             /* имя метода, указатель функции C, всегда кортеж */
  {NULL, NULL}                         /* маркер конца таблицы */
};
,
/* инициализация модуля */
void inithello()                                /* вызывается при первом импорте */
{                                               /* при динамической загрузке важно имя */
  (void) Py_InitModule("hello", hello_methods);  /* имя модуля, указатель таблицы */
}
```

В конечном итоге код Python вызовет функцию `message` из этого файла C, передав объект строки и получив его обратно. Сначала, однако, нужно как-то скомпоновать файл с интерпретатором Python. Для использования этого файла C в сценарии Python скомпилируйте его в динамически загружаемый объектный файл (например, *hello.so* в Linux) с помощью make-файла типа приведенного в примере 19.2 и поместите полученный объектный файл в каталог, присутствующий в пути поиска модулей PYTHONPATH, так же, как это делается для файлов *.py* или *.pyc*.[1]

Пример 19.2. PP2E\Integrate\Extend\Hello\makefile.hello

```
################################################################
# Компиляция hello.c в разделяемый объектный файл под Linux,
# динамически загружаемый при первом импорте в Python.
# MYPY - каталог, где находятся файлы заголовков Python.
################################################################

PY = $(MYPY)

hello.so: hello.c
    gcc hello.c -g -I$(PY)/Include -I$(PY) -fpic -shared -o hello.so

clean:
    rm -f hello.so core
```

Это make-файл для Linux (на других платформах может быть иным); чтобы собрать с его помощью модуль расширения, просто введите в оболочке `make -f makefile.hello`. Не забудьте включить путь к каталогу установки Python с помощью флагов `-I`, чтобы иметь доступ к файлам заголовков Python. При такой компиляции Python автоматически загружает и компонует модуль C при первом импорте сценарием Python.

Наконец, чтобы вызвать функцию C из программы Python, импортируйте модуль `hello` и вызовите его функцию `hello.message` со строкой:

---

[1] Так как Python при импорте всегда обыскивает текущий рабочий каталог, примеры этой главы будут выполняться из каталога, куда вы их компилируете («.»), без всяких копирований и перемещений файлов. Присутствие в PYTHONPATH более важно в крупных программах и установках.

```
[mark@toy ~/.../PP2E/Integrate/Extend/Hello]$ make -f makefile.hello

[mark@toy ~/.../PP2E/Integrate/Extend/Hello]$ python
>>> import hello                                    # импорт модуля C
>>> hello.message('world')                          # вызов функции C
'Hello, world'
>>> hello.message('extending')
'Hello, extending'
```

Вот и все – вы только что вызвали из Python функцию интегрированного модуля C. Самое важное, на что здесь нужно обратить внимание, – это то, что функция C выглядит точно так, как если бы она была написана на Python. Вызывающая программа Python отправляет и получает при вызове обычные строковые объекты; интерпретатор Python осуществляет маршрутизацию вызова функции C, а функция C сама справляется с преобразованием данных Python/C.

На самом деле `hello` вообще мало чем выделяется как модуль расширения на C, за исключением имени файла. Код Python импортирует модуль и загружает его атрибуты, как если бы тот был написан на Python. Модули расширений на C даже отвечают на вызовы `dir`, как обычно, и обладают стандартными атрибутами модуля и имени файла (хотя имя файла в этом случае не оканчивается на *.py* или *.pyc*):

```
>>> dir(hello)                                      # атрибуты модуля C
['__doc__', '__file__', '__name__', 'message']

>>> hello.__name__, hello.__file__
('hello', './hello.so')

>>> hello.message                                   # объект функции C
<built-in function message>
>>> hello                                           # объект модуля C
<module 'hello' from './hello.so'>
```

Как и для любого модуля Python, можно обращаться к расширению на C из файла сценария. Файл Python в примере 19.3 импортирует и использует модуль расширения на C.

Пример 19.3. PP2E\Integrate\Extend\Hello\hellouse.py

```
import hello

print hello.message('C')
print hello.message('module ' + hello.__file__)

for i in range(3):
    print hello.message(str(i))
```

Этот сценарий можно запускать, как всякий другой – при первом импорте модуля `hello` Python автоматически находит объектный файл *.so* модуля на C в каталоге, входящем в PYTHONPATH, и динамически включает его в процесс. Весь вывод этого сценария представляет строки, возвращаемые функцией C из файла *hello.c*:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Hello]$ python hellouse.py
Hello, C
Hello, module ./hello.so
Hello, 0
Hello, 1
Hello, 2
```

Компиляция и компоновка

Теперь, после несколько более длинного рассказа, дополним его недостающими деталями. Всегда нужно компилировать и каким-то образом компоновать файлы расширений C типа примера *hello.c* с интерпретатором Python, чтобы сделать их доступными сценариям Python, но в способах осуществления этого есть некоторая гибкость. Например, для компиляции этого файла C под Linux можно использовать и такое правило:

```
hello.so: hello.c
    gcc hello.c -c -g -fpic -I$(PY)/Include -I$(PY) -o hello.o
    gcc -shared hello.o -o hello.so
    rm -f hello.o
```

Для компиляции этого файла C в разделяемый объектный файл под Solaris можно использовать что-нибудь вроде следующего:

```
hello.so: hello.c
    cc hello.c -c -KPIC -o hello.o
    ld -G hello.o -o hello.so
    rm hello.o
```

На других платформах существуют еще большие отличия. Так как опции компиляторов сильно различаются, необходимо обратиться к документации для своего компилятора C или C++ либо к руководствам по расширениям Python, чтобы выяснить специфические для платформы и компилятора детали. Задача в том, чтобы выяснить, как компилировать исходный файл C в разделяемый или динамически загружаемый объектный файл в соответствии с принципами вашей платформы. После этого все становится просто: Python на сегодняшний день поддерживает динамическую загрузку расширений C на всех основных платформах.

Динамическое связывание

Технически то, что я вам пока демонстрировал, называется «динамическим связыванием» и представляет собой один из двух способов компоновки компилированных расширений C с интерпретатором Python. Поскольку альтернативное «статическое связывание» более сложно, почти всегда следует прибегать к динамическому связыванию. Чтобы осуществить динамическое связывание, нужно просто:

1. Компилировать *hello.c* в разделяемый объектный файл
2. Поместить объектный файл в каталог, входящий в путь поиска модулей Python

То есть после компиляции файла с исходным кодом в разделяемый объектный файл скопируйте или переместите объектный файл в каталог, перечисленный в PYTHONPATH. Он будет автоматически загружен и скомпонован интерпретатором Python во время выполнения, когда модуль впервые импортируется в каком-либо месте процесса Python (например, из интерактивного приглашения, автономной или встроенной программы Python или вызова C API).

Обратите внимание, что единственным именем, не определенным как `static` в примере *hello.c*, является функция инициализации. Python вызывает эту функцию по имени после загрузки объектного файла, поэтому ее имя должно быть глобальным в C и обычно иметь вид «initX», где «X» обозначает как имя модуля в операциях импорта Python, так и имя, передаваемое в `Py_InitModule`. Все остальные имена в файлах расширений C произвольны, потому что доступ к ним производится через указатели C, а не по именам (подробнее об этом ниже). Имя исходного файла C тоже произвольно – при импорте Python интересует только компилированный объектный файл.

Статическое связывание

При статическом связывании расширения вставляются в интерпретатор Python на постоянной основе. Однако осуществить его сложнее, потому что приходится перекомпилировать сам Python, а следовательно, иметь дистрибутив исходного кода Python (исполняемого файла интерпретатора недостаточно). Чтобы связать этот пример статически, добавьте строку

```
hello ~/PP2E/Integrate/Extend/Hello/hello.c
```

в файл конфигурации `Modules/Setup` в дереве исходного кода Python. Либо можно скопировать свой файл C в каталог `Modules` (или поместить там ссылку на него с помощью команды *ln*) и добавить в `Setup` строку типа `hello hello.c`.

После этого нужно перекомпилировать сам Python, выполнив команду *make* на верхнем уровне дерева исходного кода Python. Python перестроит собственные make-файлы, чтобы включить модуль, который вы добавили в `Setup`, и ввести ваш код в состав интерпретатора и его библиотек. В действительности разницы между расширениями C, написанными пользователями Python, и сервисами, стандартно входящими в язык, нет; Python построен с помощью этого самого интерфейса. Полный формат строки объявления модуля выглядит так (подробнее смотрите в файле конфигурации `Modules/Setup`):

```
<module> ... [<sourceOrObjectFile> ...] [<cpparg> ...] [<library> ...]
```

По этой схеме имя функции инициализации модуля должно соответствовать имени, указанному в файле `Setup`, иначе при сборке Python заново будут получены сообщения об ошибках компоновки. Имя исходного или объектного файла не обязательно должно совпадать с именем модуля; самое левое имя будет именем результирующего модуля Python.

Сравнение статического и динамического связывания

Статическое связывание действует на любой платформе и не требует дополнительного make-файла для компиляции расширений. Оно может оказаться полезным, если нежелательно поставлять расширения в виде отдельных файлов или целевая платформа не поддерживает динамическое связывание. Недостатками являются необходимость обновления файла конфигурации Python `Setup` и перекомпиляции самого интерпретатора Python, поэтому для использования статического связывания должен иметься полный дистрибутив исходного кода Python. Кроме того, статически связанные расширения всегда добавляются в интерпретатор вне зависимости от использования их в конкретной программе. В результате без нужды может увеличиться объем памяти, требуемый всем программам Python.

При динамическом связывании все равно требуются include-файлы Python, но добавлять расширения C можно даже в том случае, когда у вас есть только исполняемый двоичный файл интерпретатора Python. Так как расширения являются отдельными объектными файлами, нет необходимости перекомпилировать сам Python или иметь доступ к полному дистрибутиву исходного кода. А так как в этом режиме объектные файлы загружаются только по требованию, в целом получаются также меньшие по размеру исполняемые файлы – Python загружает в память только те расширения, которые действительно импортируются выполняемой программой. Иными словами, если платформа допускает динамическое связывание, вероятно, его и следует использовать.

Анатомия модуля расширения на C

Этот простой пример *hello.c* иллюстрирует общую структуру всех модулей на C. Она может несколько различаться, но этот файл состоит из типичного стереотипного кода:

Файлы заголовков Python

Файл C сначала включает стандартный заголовочный файл *Python.h* (из каталога установки Python `Include`). Этот файл определяет почти все имена, экспортируемые Python API в C, и служит отправной точкой для изучения самого API.

Функции методов

Затем файл определяет функцию, которая должна вызываться интерпретатором Python в ответ на обращение из программ Python. Функции C получают в качестве входных данных два объекта Python и возвращают обратно интерпретатору объект Python или `NULL`, который возбуждает в сценарии исключительную ситуацию (подробнее об этом ниже). В C `PyObject*` представляет указатель на родовой объект Python; можно использовать имена более специфических типов, но это требуется не всегда. Все функции модуля C могут быть объявлены как «static» (локальные для файла), потому что Python вызывает их по указателю, а не по имени.

Таблица регистрации

Ближе к концу файла находится инициализированная таблица (массив), отображающая *имена* функций на *указатели* функций (адреса). Имена из этой таблицы становятся именами атрибутов модуля, посредством которых код Python вызывает функции C. Указатели в этой таблице используются интерпретатором для диспетчеризации вызовов функций C. В сущности, таблица «регистрирует» атрибуты модуля. Завершает таблицу элемент `NULL`.

Функция инициализации

Наконец, в файле C есть функция инициализации, которую Python вызывает при первом импорте этого модуля в программу Python. Эта функция вызывает функцию API `Py_InitModule`, чтобы построить словарь атрибутов нового модуля по записям в таблице регистрации и создать запись для модуля на C в таблице `sys.modules` (описанной в главе 2 «Системные инструменты»). После инициализации вызовы из Python маршрутизируются прямо функции C через указатели функций в таблице регистрации.

Преобразование данных

Функции модуля на C несут ответственность за преобразование объектов Python в/из типов данных C. В примере 19.1 `message` получает два входных объекта Python, передаваемых из интерпретатора Python: `args` является кортежем Python, содержащим аргументы, переданные из вызвавшей программы Python (значения, перечисляемые в круглых скобках в программе Python), а `self` игнорируется; он используется только в расширениях *типов* (обсуждаемых ниже в этой главе).

Закончив свои дела, функция C может возвратить интерпретатору Python следующее: объект Python (известный в C как `PyObject*`) в качестве действительного результата; значение Python `None` (известное в C как `Py_None`), если функция не возвращает действительного результата; `NULL`-указатель C, чтобы обозначить ошибку и возбудить исключительную ситуацию Python.

Есть отдельные средства API для обработки входных преобразований (Python в C) и выходных преобразований (C в Python). Функции C должны реализовывать свои сигнатуры вызова (списки и типы аргументов), надлежащим образом используя эти средства.

Из Python в C: использование списков аргументов Python

Когда выполняется функция C, переданные из сценария Python аргументы доступны через объект кортежа Python `args`. Функция API `PyArg_Parse` (и родственная ей

PyArg_ParseTuple, предполагающая преобразование объекта кортежа) предоставляет простейший способ извлечения и преобразования переданных аргументов в формат C.

PyArg_Parse принимает объект Python, строку формата и список переменной длины целевых адресов C. Она преобразует элементы кортежа к значениям типов данных C в соответствии со строкой формата и сохраняет результаты в переменных C, адреса которых ей переданы. Ее работа похожа на действие функции C scanf. Например, модуль hello преобразует передаваемый ему аргумент строки Python в тип C char* с помощью кода преобразования s:

```
PyArg_Parse(args, "(s)", &fromPython)      # или PyArg_ParseTuple(args, "s",...
```

Для обработки нескольких аргументов коды форматов соединяются в одну строку, и для каждого кода в строке указывается соответствующий целевой адрес C. Например, чтобы преобразовать список аргументов, содержащий строку, целое и еще одну строку в C, используется вызов:

```
PyArg_Parse(args, "(sis)", &s1, &i, &s2)   # или PyArg_ParseTuple(args, "sis",...
```

Для проверки отсутствия переданных аргументов используется пустая строка формата, например: PyArg_Parse(args, "()"). Этот вызов API проверяет соответствие числа и типов аргументов, переданных из Python, строке формата в вызове. При несоответствии устанавливается исключительная ситуация и возвращается ноль в C (подробнее об ошибках ниже).

Из Python в C: использование значений, возвращаемых Python

Как будет ясно из главы 20 «Встраиваем Python», функции API могут также возвращать в C в качестве результата объекты Python, когда Python выполняется как встроенный язык. Преобразование возвращаемых Python значений в этом режиме выполняется почти так же, как преобразование аргументов Python, передаваемых функциям расширения C, за исключением того, что возвращаемые Python значения не всегда являются кортежами. Чтобы преобразовать возвращаемые Python объекты в формат C, используется PyArg_Parse. В отличие от PyArg_ParseTuple, этот вызов принимает такие же аргументы, но не ожидает, что объект Python будет кортежем.

Из C в Python: возврат значений в Python

Есть два способа преобразования данных C в объекты Python: с помощью функций API для конкретных типов или общей функции построения объекта Py_BuildValue. Последний способ является более общим и, по существу, обращает PyArg_Parse, поскольку Py_BuildValue преобразует данные C в объекты Python согласно строке формата. Например, чтобы создать объект строки Python из C char*, модуль hello использует код преобразования s:

```
return Py_BuildValue("s", result)          # "result" является C char []/*
```

Вместо этого можно использовать более специфические конструкторы объектов:

```
return PyString_FromString(result)         # тот же результат
```

Оба вызова создают объект строки Python из указателя на массив символов C. Полный список таких вызовов можно найти в стандартных теперь руководствах Python по API расширений и времени выполнения. Но Py_BuildValue проще запомнить, а его синтаксис позволяет создавать списки за один шаг, что описывается ниже.

Стандартные коды преобразования

За малым исключением, PyArg_Parse(Tuple) и Py_BuildValue используют одинаковые коды в строках формата. Список всех поддерживаемых кодов преобразований есть в

руководствах по расширению Python. Наиболее часто используемые из них приведены в табл. 19.1; форматы кортежа, списка и словаря могут быть вложенными.

Таблица 19.1. Частые коды преобразования данных Python/C

Код строки формата	Тип данных C	Тип объекта Python
`s`	`char*`	Строка
`s#`	`char*, int`	Строка, длина
`i`	`Int`	Целое
`l`	`long int`	Целое
`c`	`Char`	Строка
`f`	`Float`	С плавающей точкой
`d`	`Double`	С плавающей точкой
`O`	`PyObject*`	Исходный (не преобразованный) объект
`O&`	`&converter, void*`	Преобразованный объект (вызывает конвертер)
(*элементы*)	Цели или значения	Вложенный кортеж
[*элементы*]	Серия аргументов/значений	Список
{*элементы*}	Серия аргументов `key,value`	Словарь

Эти коды в основном интуитивны (например, `i` осуществляет преобразование между C `int` и объектом Python integer), однако вот несколько замечаний по использованию этой таблицы:

- Преобразуя *в* C, передавайте адрес `char*` для кодов `s`, а не адрес массива `char`: Python копирует адрес существующей строки C (а ее нужно копировать, чтобы хранить неограниченно в C: используйте `strdup`).
- Код `O` полезен для передачи необработанных объектов Python между языками; когда есть указатель на необработанный объект, можно воспользоваться средствами API низкого уровня для обращения к атрибутам объекта по имени, индексировать и делать срезы последовательностей, и т. д.
- Код `O&` позволяет передавать функции-конвертеры C для специальных преобразований. Это оказывается удобным при специальной обработке, чтобы отобразить объект в тип данных C, не поддерживаемый непосредственно кодами преобразований (например, для преобразования целых структур C или экземпляров классов C++). Подробности смотрите в руководстве по расширениям.
- Последние две строки, `[...]` и `{...}`, в настоящее время поддерживаются только `Py_BuildValue`: можно строить списки и словари с помощью строк формата, но нельзя распаковывать их. Взамен в API есть специфические функции для доступа к компонентам последовательности и отображения по указателю на объект.

`PyArg_Parse` поддерживает некоторые другие коды, которые не должны вкладываться в форматы кортежей (`(...)`):

- `|` Остальные аргументы являются необязательными (*varargs*). Целевые адреса C остаются без изменений, если в кортеже Python отсутствуют аргументы. Например, `si|sd` требует два аргумента, но допускает до четырех.

: Далее идет имя функции, используемой в сообщениях об ошибках, устанавливаемых вызовом (несоответствие аргументов). Обычно Python устанавливает сообщение об ошибке как общую строку.

; Далее следует полное сообщение об ошибках, идущее до конца строки формата.

Этот список кодов форматов неполон, а набор кодов преобразования может быть в будущем расширен; детали смотрите в руководстве по расширению Python.

Обработка ошибок

При создании расширений на C нужно знать об ошибках, которые могут произойти по обе стороны этого языкового барьера. Следующие разделы касаются обеих возможностей.

Возбуждение исключительных ситуаций Python в C

Функции модуля расширений на C возвращают как результирующий объект значение C `NULL`, чтобы сигнализировать об ошибке. Когда управление возвращается в Python, результат `NULL` возбуждает обычную исключительную ситуацию Python в коде Python, вызвавшем функцию C. Чтобы указать исключительную ситуацию, код C может также установить тип и дополнительные данные для возбуждаемых им исключительных ситуаций. Например, функция API `PyErr_SetString` устанавливает объект исключительной ситуации равным объекту Python и задает дополнительные данные исключительной ситуации как строку символов:

```
PyErr_SetString(ErrorObject, message)
```

Мы воспользуемся этим в следующем примере, чтобы уточнить исключительные ситуации, возбуждаемые, когда C обнаруживает ошибку. Модули на C могут также устанавливать встроенные исключительные ситуации Python; например, возвращение `NULL` после

```
PyErr_SetString(PyExc_IndexError, "index out-of-bounds")
```

возбуждает стандартную исключительную ситуацию Python `IndexError` со строкой данных сообщения. Когда ошибка возбуждается внутри функции Python API, Python автоматически устанавливает объект исключительной ситуации и связанные с ним «дополнительные данные»; нет необходимости устанавливать его снова в вызывающей функции C. Например, когда в функции `PyArg_Parse` обнаруживается ошибка передачи аргументов, модуль стека `hello` просто возвращает `NULL`, чтобы распространить исключительную ситуацию в охватывающий слой Python, а не устанавливать собственное сообщение.

Обнаружение ошибок, возникающих в Python

Функции API Python могут вызываться из функций расширений на C или из охватывающего слоя C, когда Python встраивается. В том и другом случае вызывающий код C просто проверяет возвращаемое значение, чтобы обнаружить ошибки, возбуждаемые в функциях API Python. Для функций, результатом которых является указатель, Python в случае ошибок возвращает указатель `NULL`. Для функций с целочисленным результатом Python обычно возвращает код состояния −1 для сообщения об ошибке и 0 или положительное значение в случае успеха. (`PyArg_Parse` составляет исключение из этого правила: она возвращает 0, обнаружив ошибку.) Чтобы сделать программу устойчивой, нужно проверять возвращаемые коды на наличие ошибки после большинства вызовов API Python; некоторые вызовы могут быть неуспешными по неожиданным причинам (например, переполнение памяти).

Счетчики ссылок

Интерпретатор Python использует схему подсчета ссылок для осуществления уборки мусора. В каждом объекте Python ведет подсчет числа мест, из которых к нему было обращение; когда этот счетчик достигает нуля, Python автоматически забирает себе пространство памяти объекта. Обычно Python управляет счетчиками ссылок на объекты невидимым образом; программы Python создают и используют объекты, не заботясь об управлении пространством памяти.

Однако при расширении и встраивании Python интегрированный код C отвечает за управление счетчиками ссылок используемых им объектов Python. Насколько это важно, зависит от того, сколько чистых объектов Python обрабатывается модулем C и какие функции API Python он вызывает. В простых программах счетчики ссылок не представляют большого интереса; модуль `hello`, например, вообще не осуществляет вызовов, управляющих подсчетом ссылок.

Однако если API используется интенсивно, эта задача может стать важной. В последующих примерах появятся вызовы такого вида:

- `Py_INCREF(obj)` инкрементирует счетчик ссылок объекта.
- `Py_DECREF(obj)` инкрементирует счетчик ссылок объекта (освобождает память, если ноль).
- `Py_XINCREF(obj)` аналогичен `Py_INCREF(obj)`, но игнорирует указатель объекта `NULL`.
- `Py_XDECREF(obj)` аналогичен `py_DECREF(obj)`, но игнорирует указатель объекта `NULL`.

Предполагается, что функции C возвращают либо объект с инкрементированным счетчиком ссылок, либо `NULL`, сигнализируя об ошибке. По общему правилу функции API, которые создают новые объекты, инкрементируют свои счетчики ссылок перед тем, как возвратить их в C; если только новый объект не должен быть передан обратно в Python, создающая его программа C должна в конечном счете декрементировать счетчик объекта. При расширении дело относительно просто: счетчик ссылок объекта аргумента не требуется декрементировать, а новые результирующие объекты возвращаются в Python с нетронутыми счетчиками ссылок.

Оборотной стороной счетчика ссылок является то, что Python никогда не уберет объект, удерживаемый C, если C инкрементирует счетчик ссылок объекта (или не декрементирует счетчик объекта, которым владеет). Несмотря на необходимость вызовов, управляющих счетчиком, схема Python уборки мусора довольно хорошо подходит для интеграции с C.

SWIG – генератор интегрирующего кода

Но не делайте этого. Я показываю основы расширений C для того, чтобы вам стала понятна лежащая в основе структура, но сегодня расширения C обычно лучше и проще реализуются с помощью генератора интегрирующего кода SWIG.

SWIG – Simplified Wrapper and Interface Generator (упрощенный генератор оболочек и интерфейсов) является системой open source, созданной Дэйвом Бизли (Dave Beazley). Она использует объявления типов C и C++ для генерации законченных модулей расширения C, интегрирующих существующие библиотеки для использования в сценариях Python. Генерируемые модули расширения C являются законченными: они автоматически осуществляют преобразование данных, протоколы ошибок, управление счетчиками ссылок и прочее.

То есть SWIG автоматически генерирует весь «связующий» код, необходимый для вставки компонентов C и C++ в программы Python. Нужно скомпилировать его

вывод, и на этом создание расширения завершается. Вам остается разобраться с деталями компиляции и компоновки, а остальную часть работы, связанной с созданием расширения на C, выполняет SWIG.

Простой пример SWIG

Например, вместо всего этого кода C из предыдущего раздела вы пишете функцию C, которую нужно использовать из Python, без всякой логики интеграции с Python, как если бы ее требовалось вызывать только из C. Это иллюстрирует пример 19.4.

Пример 19.4. PP2E\Integrate\Extend\HelloLib\hellolib.c

```
/*****************************************************************************
 * Файл простой библиотеки C с одной функцией "message", которая должна стать
 * доступной в программах Python. Здесь ничто не говорит о Python--эта функция C
 * может вызываться из программы C, как и из Python (с помощью связующего кода).
 *****************************************************************************/

#include <string.h>
#include <hellolib.h>

static char result[64];                  /* не экспортируется */

char *
message(char *label)                     /* экспортируется */
{
    strcpy(result, "Hello, ");           /* создать строку C */
    strcat(result, label);               /* добавить переданную метку */
    return result;                       /* возвратить временную переменную */
}
```

Раз уж на то пошло, определите обычный файл заголовков C и объявите эту функцию внешним образом, как показано в примере 19.5. Это может быть излишеством, но подчеркивает суть.

Пример 19.5. PP2E\Integrate\Extend\HelloLib\hellolib.h

```
/*****************************************************************************
 * Определяет экспорт hellolib.c в пространство имен C, а не в программы
 * Python--последнее определяется таблицей регистрации
 * методов в коде модуля расширения Python, а не данным файлом .h;
 *****************************************************************************/

extern char *message(char *label);
```

Теперь вместо всего связующего кода расширения Python, показанного в предыдущем разделе, напишите входной файл объявлений типов SWIG, как в примере 19.6.

Пример 19.6. PP2E\Integrate\Extend\Swig\hellolib.i

```
/*************************************************
 * Файл описания модуля Swig для файла библиотеки C.
 * Генерация по команде "swig -python hellolib.i".
 *************************************************/

%module hellowrap

%{
#include <hellolib.h>
%}
```

```
extern char *message(char*);     /* или: %include "../HelloLib/hellolib.h"   */
                                 /* или: %include hellolib.h и использовать -I arg */
```

Этот файл раскрывает сигнатуру типа функции C. Вообще говоря, SWIG сканирует файлы, содержащие объявления ANSI C и C++. Его входной файл может быть файлом описания интерфейса (обычно с суффиксом *.i*) или файлом заголовков, или исходного кода C/C++. Формат таких файлов интерфейсов, как этот, чаще всего используется для ввода; они могут содержать *комментарии* в формате C или C++, *объявления* типов, как в стандартных файлах заголовков, и *директивы* SWIG, которые начинаются с `%`. Например:

- `%module` задает имя модуля, которое будет использоваться при импорте в Python.
- `%{...%}` содержит код, добавляемый в создаваемый файл оболочки дословно.
- `extern` утверждения объявляют экспорт в обычном синтаксисе ANSI C/C++.
- `%include` заставляет SWIG сканировать другой файл (флаги `-I` задают пути поиска).

В этом примере можно было заставить SWIG прочесть файл заголовков *hellolib.h* непосредственно. Но одним из преимуществ написания специальных входных файлов SWIG типа *hellolib.i* является то, что можно отобрать функции, которые будут заключены в оболочку и экспортированы в Python; просмотр всего файла заголовков библиотеки создает оболочку для всего, что в нем определено.

В действительности SWIG является утилитой, которая запускается из сценариев компиляции, а не языком программирования, поэтому здесь особенно нечего больше показывать. Просто добавьте в свой make-файл запуск SWIG и скомпилируйте его вывод для компоновки с Python. Пример 19.7 показывает способ осуществления этого под Linux.

Пример 19.7. PP2E\Integrate\Extend\Swig\makefile.hellolib-swig

```
################################################################
# Использование SWIG для интеграции hellolib.c с целью использования в программах Python.
################################################################

# если вы не выполнили make install
SWIG = ./myswig

PY   = $(MYPY)
LIB  = ../HelloLib

# библиотека плюс ее оболочка
hellowrap.so: hellolib_wrap.o $(LIB)/hellolib.o
    ld -shared hellolib_wrap.o $(LIB)/hellolib.o -o hellowrap.so

# генерируемый код модуля оболочки
hellolib_wrap.o: hellolib_wrap.c $(LIB)/hellolib.h
    gcc hellolib_wrap.c -c -g -I$(LIB) -I$(PY)/Include -I$(PY)

hellolib_wrap.c: hellolib.i
    $(SWIG) -python -I$(LIB) hellolib.i

# код библиотеки C (в другом каталоге)
$(LIB)/hellolib.o: $(LIB)/hellolib.c $(LIB)/hellolib.h
    gcc $(LIB)/hellolib.c -c -g -I$(LIB) -o $(LIB)/hellolib.o

clean:
    rm -f *.o *.so core
force:
    rm -f *.o *.so core hellolib_wrap.c hellolib_wrap.doc
```

При выполнении этого make-файла с входным файлом *hellolob.i* SWIG создает два файла:

- *hellolib_wrap.doc* является текстовой сводкой функций в модуле.
- *hellolib_wrap.c* представляет связующий код модуля расширения на C.[1]

Этот make-файл просто запускает SWIG, компилирует сгенерированный файл связующего кода C в объектный файл *.o*, а затем соединяет его с компилированным объектным файлом *hellolib.c*, создавая *hellowrap.so*. Последний служит динамически загружаемым модулем расширения C, который должен быть помещен в каталог, входящий в путь поиска модулей Python (или «.» при работе в том же каталоге, куда производится компиляция).

Если SWIG готов к работе, запустите make-файл, чтобы создать и откомпилировать оболочки для функции C. Вот как идет процесс компиляции под Linux:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Swig]$ make -f makefile.hellolib-swig
./myswig -python -I../HelloLib hellolib.i
Generating wrappers for Python
gcc hellolib_wrap.c -c -g -I../HelloLib ...здесь удалена часть текста...
ld -shared hellolib_wrap.o ../HelloLib/hellolib.o -o hellowrap.so
```

И после выполнения этого make-файла все готово. Сгенерированный модуль на C используется точно так же, как написанная вручную версия, причем SWIG автоматически позаботился о сложных местах:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Swig]$ python
>>> import hellowrap                          # импорт файла связка+библиотека
>>> hellowrap.__file__                        # импорт файла связка+библиотека
'./hellowrap.so'
>>> hellowrap.message('swig world')
'Hello, swig world'
```

Иными словами, научившись пользоваться SWIG, можно в целом забыть детали кодирования интеграции, о которых рассказывалось в этой главе. На самом деле SWIG настолько сведущ в генерации связующего кода Python, что обычно значительно проще и менее чревато ошибками писать C-расширения для Python сначала как чистые библиотеки C или C++, а затем добавлять их в Python, пропуская файлы заголовков через SWIG, как продемонстрировано здесь.

Подробности о SWIG

Конечно, чтобы запустить SWIG, нужно его иметь: он не входит в сам Python. Если его нет у вас на машине, загрузите SWIG из веб (или возьмите с прилагаемого к книге CD) и скомпилируйте из исходного кода. Вам понадобится компилятор C++ (например, `g++`), но установка очень проста; подробности в файле README. SWIG является программой командной строки и обычно может быть запущен просто строкой:

```
swig -python hellolib.i
```

В моей среде компиляции несколько сложнее, потому что у меня установлена индивидуальная сборка SWIG. Я запускаю SWIG из такого сценария `csh` с именем `myswig`:

---

[1] При желании можете разобрать этот сгенерированный файл с CD, прилагаемого к книге. Смотрите также на этом CD файл *PP2E\Integrate\Extend\HelloLib\hellolib_wrapper.c* с написанным вручную эквивалентом; последний короче, потому что SWIG генерирует также дополнительный вспомогательный код.

```
#!/bin/csh
# запуск индивидуально собранного swig

source $PP2EHOME/Integrate/Extend/Swig/setup-swig.csh
swig $*
```

Этот файл, в свою очередь, задает указатели на каталог установки SWIG, загружая следующий файл `csh` с именем *setup-swig.csh*:

```
# source в csh для запуска индивидуальной установки SWIG

setenv SWIG_LIB /home/mark/PP2ndEd/dev/examples/SWIG/SWIG1.1p5/swig_lib
alias swig ~/home/mark/PP2ndEd/dev/examples/SWIG/SWIG1.1p5/swig~
```

Но эти файлы не понадобятся вам, если выполнить команду `make install` в каталоге исходного кода SWIG, которая разместит его по стандартным адресам.

Попутно в этой главе я покажу вам еще несколько основанных на SWIG вариантов в оставшихся примерах. За полными сведениями следует обратиться к руководству Python по SWIG, однако бегло взглянем еще на некоторые важные моменты в SWIG:

«Теневые» классы C++

Далее в этой главе я покажу, как использовать SWIG, чтобы интегрировать классы C++ для использования в сценариях Python. Исходя из объявлений классов C++ SWIG генерирует связующий код, благодаря которому классы C++ выглядят в сценариях Python как классы Python. В действительности классы C++ и становятся классами Python с помощью SWIG; получается то, что в SWIG называется «теневым» классом C++, который взаимодействует с написанным на C++ модулем расширения, в свою очередь взаимодействующим с классами C++. Так как внешним уровнем интеграции являются классы Python, для них можно создавать в Python подклассы и обрабатывать их экземпляры с помощью обычного синтаксиса объектов Python.

Переменные

Помимо функций и классов C++ SWIG может делать оболочки глобальных переменных и констант C для использования в Python: они становятся атрибутами объекта с именем `cvar`, вставляемого в генерируемые модули (например, `module.cvar.name` загружает значение переменной C `name` из сгенерированного SWIG модуля оболочки).

Указатели

Для единообразия, а также возможности проверки безопасности типов SWIG пересылает указатели из одного языка в другой в виде строк (а не специальных типов Python). Например, указатель на тип `Vector` может иметь вид `_100f8e2_Vector_p`. Обычно это не важно, потому что и в C смотреть на значения указателей не очень интересно. Можно также заставить SWIG обрабатывать выходные параметры и ссылки C++.

Структуры

Конструкции C `struct` преобразуются в набор функций доступа `get` и `set`, которые вызываются при получении и присваивании значений полям через указатель объекта `struct` (например, `module.Vector_fieldx_get(v)` получает значение C `Vector_fieldx` из `v` – указателя на `Vector`, как `v->fieldx` в C). Аналогичные функции доступа генерируются для членов-данных и методов классов C++ (в C++ `class` похож на `struct` с дополнительным синтаксисом), но наличие в SWIG теневых классов позволяет обращаться с помещенными в оболочку классами, как с классами Python, а не вызывать функции доступа низкого уровня.

Хотя примеры SWIG в этой книге просты, следует знать, что с такой же легкостью SWIG обрабатывает промышленные библиотеки. Например, разработчики Python успешно использовали SWIG для интеграции таких сложных библиотек, как расширения Windows и часто используемые графические API.

SWIG может также генерировать интегрирующий код для других языков сценариев, таких как Tcl и Perl. В действительности одна из основных его целей заключается в создании компонентов, независимых от выбора языка сценария – библиотеки C/C++ можно включать в те языки сценариев, которые вы предпочитаете использовать (я предпочитаю Python, но могу быть не беспристрастен). Поддержка в SWIG таких вещей, как классы, лучше всего организована, по-видимому, для Python, возможно, потому, что Python считается сильным по части классов. Как средство интеграции, независимое от языка, SWIG отчасти решает те же проблемы, что и такие системы, как COM и CORBA (описанные в главе 20), но его решение основано на генерации кода, а не на объектной модели.

SWIG можно взять с прилагаемого к книге CD или с домашней страницы в веб *http://www.swig.org*. Помимо полного исходного кода вместе с SWIG поставляется замечательная документация (в том числе руководство специально для Python), поэтому я не стану рассказывать в этой книге обо всех его возможностях. В документации также рассказано, как строить расширения SWIG в Windows. Есть данные, что книга по SWIG находится в настоящее время в работе, поэтому проверяйте список книг на *http://www.python.org*.

Создание оболочек для вызовов окружения C

Перейдем к более практическому приложению модулей расширения C. Написанный вручную файл C в примере 19.8 интегрирует вызовы переменных окружения оболочки стандартной библиотеки C `getenv` и `putenv` для использования в сценариях Python.

Пример 19.8. PP2E\Integrate\Extend\CEnviron\cenviron.c

```
/*************************************************************
 * Модуль расширения на C для Python с именем "cenviron". Служит оболочкой
 * подпрограмм библиотеки C getenv/putenv, позволяющей использовать их в Python.
 *************************************************************/

#include <Python.h>
#include <stdlib.h>
#include <string.h>

/***********************/
/* 1) функции модуля */
/***********************/

static PyObject *                                   /* возвращает объект */
wrap_getenv(PyObject *self, PyObject *args)         /* self не используется */
{                                                   /* аргументы из python */
    char *varName, *varValue;
    PyObject *returnObj = NULL;                         /* null=exception */

    if (PyArg_Parse(args, "s", &varName)) {             /* Python -> C */
        varValue = getenv(varName);                     /* вызов C getenv */
        if (varValue != NULL)
            returnObj = Py_BuildValue("s", varValue);   /* C -> Python */
        else
            PyErr_SetString(PyExc_SystemError, "Error calling getenv");
    }
```

```
    return returnObj;
}

static PyObject *
wrap_putenv(PyObject *self, PyObject *args)
{
    char *varName, *varValue, *varAssign;
    PyObject *returnObj = NULL;

    if (PyArg_Parse(args, "(ss)", &varName, &varValue))
    {
        varAssign = malloc(strlen(varName) + strlen(varValue) + 2);
        sprintf(varAssign, "%s=%s", varName, varValue);
        if (putenv(varAssign) == 0) {
            Py_INCREF(Py_None);                    /* успешный вызов C */
            returnObj = Py_None;                   /* ссылка на None */
        }
        else
            PyErr_SetString(PyExc_SystemError, "Error calling putenv");
    }
    return returnObj;
}

/*************************/
/* 2) таблица регистрации */
/*************************/

static struct PyMethodDef cenviron_methods[] = {
    {"getenv", wrap_getenv},
    {"putenv", wrap_putenv},        /* имя, адрес метода */
    {NULL, NULL}
};

/************************/
/* 3) инициализатор модуля */
/************************/

void initcenviron()                  /* вызывается при первом импорте */
{
    (void) Py_InitModule("cenviron", cenviron_methods);   /*имя, таблица модуля */
}
```

Сейчас этот пример менее полезен, чем тогда, когда он был включен в первое издание этой книги, – как мы узнали из части I «Системные интерфейсы», можно не только получать переменные системного окружения из таблицы `os.environ`, но и, присваивая по ключу в этой таблице, автоматически вызывать функцию C `putenv`, чтобы экспортировать новое значение в слой кода C. То есть `os.environ['key']` загружает значение переменной окружения `'key'`, а `os.environ['key']=value` присваивает значение переменной как в Python, так и в C.

Второе действие – присвоение в C – было добавлено в Python после выхода первого издания книги. Кроме дополнительного показа техники написания расширений этот пример все же служит практической цели: даже сегодня изменения переменных в коде, соединенном с процессом Python, не видны при обращении к `os.environ` в коде Python. То есть после начала программы `os.environ` отражает только последующие изменения, сделанные в коде Python.

Если требуется, чтобы код Python действительно был интегрирован с настройками оболочки, осуществляемыми в коде модулей расширения на C, необходимо применять средства обращения к окружению из библиотеки C: к `putenv` можно обращаться

как к os.putenv, но getenv отсутствует в библиотеке Python. Едва ли из-за этого могут возникнуть проблемы; но данный пример модуля расширения на C не совсем бесполезен (по крайней мере, пока Гвидо снова не закрутит гайки).[1]

Этот файл *cenviron.c* создает модуль Python с именем cenviron, который идет немного дальше, чем последний пример: он экспортирует две функции, явно задает некоторые описания исключительных ситуаций и обращается к счетчику ссылок объекта Python None (он не создается заново, поэтому необходимо добавить ссылку перед отправкой его в Python). Как и раньше, чтобы добавить этот код в Python, откомпилируйте его в объектный файл; make-файл Linux в примере 19.9 предназначен для сборки исходного кода C для динамического связывания.

Пример 19.9. PP2E\Integrate\Extend\Cenviron\makefile.cenviron

```
#################################################################
# Компилировать cenviron.c в cenviron.so--разделяемый объектный файл
# в Linux, динамически загружаемый при первом импорте.
#################################################################

PY = $(MYPY)

cenviron.so: cenviron.c
    gcc cenviron.c -g -I$(PY)/Include -I$(PY) -fpic -shared -o cenviron.so

clean:
    rm -f *.pyc cenviron.so
```

Для сборки введите в оболочке make -f makefile.cenviron. Для выполнения файл *.so* должен быть в каталоге, входящем в путь поиска Python («.» тоже годится):

```
[mark@toy ~/.../PP2E/Integrate/Extend/Cenviron]$ python
>>> import cenviron
>>> cenviron.getenv('USER')                # как os.environ[key], но загружается заново
'mark'
>>> cenviron.putenv('USER', 'gilligan')    # как os.environ[key]=value'
>>> cenviron.getenv('USER')                # C тоже видит изменения
'gilligan'
```

Как и раньше, cenviron является после импорта настоящим объектом модуля Python со всей прикрепляемой обычно информацией:

```
>>> dir(cenviron)
['__doc__', '__file__', '__name__', 'getenv', 'putenv']
>>> cenviron.__file__
'./cenviron.so'
>>> cenviron.__name__
'cenviron'
>>> cenviron.getenv
<built-in function getenv>
>>> cenviron
<module 'cenviron' from './cenviron.so'>
```

[1] Этот код тоже можно индивидуально настраивать (например, он может ограничить набор читаемых и записываемых переменных оболочки путем проверки их имен), но того же самого можно достичь с помощью оболочки для os.environ. На самом деле, поскольку os.environ является просто подклассом UserDict, который заранее загружает переменные оболочки при начальном запуске, можно *почти* добавить требуемый вызов getenv для загрузки изменений в слое C, обернув обращения os.environ в класс Python, в котором __getitem__ вызывает getenv, прежде чем передать обращение os.environ. Но вначале нужен вызов C getenv, а в os его сегодня нет.

```
>>> print cenviron.getenv('HOST'), cenviron.getenv('DISPLAY')
toy :0.0
```

Вот пример задачи, которая решается с помощью этого модуля (но необходимо, чтобы вызовы `getenv` осуществлялись присоединенным кодом C, а не Python):

```
>>> import os
>>> os.environ['USER']                      # инициализация в оболочке
'skipper'
>>> from cenviron import getenv, putenv     # прямое обращение к вызовам библиотеки C
>>> getenv('USER')
'skipper'
>>> putenv('USER', 'gilligan')              # изменения для C, а не Python
>>> getenv('USER')
'gilligan'
>>> os.environ['USER']                      # ох! - не загружает значения снова
'skipper'
```

В настоящей версии модуль расширения на C экспортирует интерфейс, основанный на функциях, но можно заключить его функции в код Python, который придаст интерфейсу любой желаемый вид. Например, в примере 19.10 доступ к функциям осуществляется через словарь и интегрируется с объектом `os.environ`.

Пример 19.10. PP2E\Integrate\Extend\Cenviron\envmap.py

```
import os
from cenviron import getenv, putenv       # получить методы модуля C

class EnvMapping:                         # заключить в класс Python
    def __setitem__(self, key, value):
        os.environ[key] = value           # при записи: Env[key]=value
        putenv(key, value)                # поместить также в os.environ

    def __getitem__(self, key):
        value = getenv(key)               # при чтении: Env[key]
        os.environ[key] = value           # проверка целостности
        return value

Env = EnvMapping()                        # создать один экземпляр
```

А пример 19.11 экспортирует функции как квалифицированные имена атрибутов вместо вызовов. Смысл здесь в том, что можно прививать много различных моделей интерфейсов к функциям расширений с помощью оболочек Python (к этой мысли мы еще вернемся, когда будем разбирать оболочки типов и теневые классы SWIG далее в этой главе).

Пример 19.11. PP2E\Integrate\Extend\Cenviron\envattr.py

```
import os
from cenviron import getenv, putenv       # получить методы модуля C

class EnvWrapper:                         # заключить в класс Python
    def __setattr__(self, name, value):
        os.environ[name] = value          # при записи: Env.name=value
        putenv(name, value)               # поместить также в os.environ

    def __getattr__(self, name):
        value = getenv(name)              # при чтении: Env.name
        os.environ[name] = value          # проверка целостности
        return value

Env = EnvWrapper()                        # создать один экземпляр
```

Но не делайте этого – SWIG

Можно вручную писать модули расширения, как мы это только что сделали, но делать это необязательно. Поскольку этот пример в действительности просто создает оболочку имеющихся в стандартных библиотеках C функций, весь файл кода C *cenviron.c* примера 19.8 можно заменить простым входным файлом SWIG, который выглядит так:

Пример 19.12. PP2E\Integrate\Extend\Swig\Environ\environ.i

```
/*************************************************************
 * Файл описания модуля Swig для генерации всего кода оболочки
 * Python для вызовов библиотеки C getenv/putenv: "swig -python environ.i".
 *************************************************************/

%module environ

%{
#include <stdlib.h>
%}

extern char * getenv(const char *varname);
extern int    putenv(const char *assignment);
```

И все готово. Ну, почти: все же нужно еще пропустить этот файл через SWIG и скомпилировать вывод. Как и прежде, просто добавьте в свой make-файл строки для SWIG, скомпилируйте результат в разделяемый объект и получите удовольствие. Пример 19.13 представляет make-файл для Linux, который это делает.

Пример 19.13. PP2E\Integrate\Extend\Swig\Environ\makefile.environ-swig

```
# компилировать расширение environ.so из кода, сгенерированного SWIG

# если только вы не выполнили make install
SWIG = ../myswig
PY   = $(MYPY)

environ.so: environ_wrap.c
    gcc environ_wrap.c -g -I$(PY)/Include -I$(PY) -shared -o environ.so

environ_wrap.c: environ.i
    $(SWIG) -python environ.i

clean:
    rm -f *.o *.so core
force:
     rm -f *.o *.so core environ_wrap.c environ_wrap.doc
```

При выполнении с *environ.i* SWIG генерирует два файла – *environ_wrap.doc* (перечень описаний функций-оболочек) и *environ_wrap.c* (файл модуля связующего кода). Так как функции, для которых здесь создаются оболочки, находятся в стандартных присоединяемых библиотеках C, объединять с генерируемым кодом нечего; этот make-файл просто запускает SWIG и компилирует файл оболочки в модуль расширения C, готовый к импорту:

```
[mark@toy ~/...../Integrate/Extend/Swig/Environ]$ make -f makefile.environ-swig
../myswig -python environ.i
Generating wrappers for Python
gcc environ_wrap.c -g -I/...еще... -shared -o environ.so
```

А вот теперь действительно все. Полученный модуль расширения C связывается при импорте и используется как прежде (за исключением того, что SWIG обработал все детали):

```
[mark@toy ~/...../Integrate/Extend/Swig/Environ]$ python
>>> import environ
>>> environ.getenv('USER')
'mark'
>>> environ.putenv('USER=gilligan')  # теперь используется схема вызова библиотеки C
0
>>> environ.getenv('USER')
'gilligan'

>>> dir(environ)
['__doc__', '__file__', '__name__', 'getenv', 'putenv']
>>> environ.__name__, environ.__file__, environ
('environ', './environ.so', <module 'environ' from './environ.so'>)
```

Можно также выполнить SWIG с файлом заголовков C, в котором определены `getenv` и `putenv`, но в результате этого оболочки будут созданы для всех функций в файле заголовков. С помощью входного файла, который приведен здесь, оболочки будут созданы только для двух библиотечных функций.

Стек строк модуля расширения на C

Поднимемся еще на одну отметку вверх – следующий модуль расширения C реализует стек строк, который можно использовать в сценариях Python. Пример 19.14 демонстрирует дополнительные вызовы API, а также служит основой для сравнения. Он примерно эквивалентен модулю стека Python, с которым мы встречались ранее в главе 17 «Структуры данных», но хранит только строки (а не произвольные объекты), ограничивает размеры хранимых строк и стека и написан на C.

К сожалению, последнее обстоятельство обусловливает сложный листинг программы – код C никогда не выглядит так же приятно, как эквивалентный код Python. В C нужно объявлять переменные, управлять памятью, реализовывать структуры данных и писать массу дополнительного синтаксиса. Если только вы не принадлежите к числу больших поклонников C, следует сосредоточиться на коде интерфейса к Python в этом файле, а не на внутреннем устройстве его функций.

Пример 19.14. PP2E\Integrate\Extend\Stacks\stackmod.c

```
/*************************************************************
 * stackmod.c: совместно используемый стек символьных строк;
 * модуль расширения на C для использования в программах Python;
 * компонуется с библиотеками python или загружается при импорте;
 *************************************************************/

#include "Python.h"             /* файлы заголовков Python */
#include <stdio.h>              /* файлы заголовков C */
#include <string.h>

static PyObject *ErrorObject;   /* локально возбуждаемые исключительные ситуации */

#define onError(message) \
       { PyErr_SetString(ErrorObject, message); return NULL; }

/*****************************************************************************
* ЛОКАЛЬНЫЕ ЛОГИКА/ДАННЫЕ (СТЕК)
*****************************************************************************/

#define MAXCHARS 2048
#define MAXSTACK MAXCHARS
```

```
static int  top = 0;                /* индекс для 'stack' */
static int  len = 0;                /* размер 'strings' */
static char *stack[MAXSTACK];       /* указатели на 'strings' */
static char strings[MAXCHARS];      /* память для хранения строк */

/*****************************************************************************
* ЭКСПОРТИРУЕМЫЕ МОДУЛЕМ МЕТОДЫ/ФУНКЦИИ
*****************************************************************************/

static PyObject *
stack_push(PyObject *self, PyObject *args)      /* args: (string) */
{
    char *pstr;
    if (!PyArg_ParseTuple(args, "s", &pstr))    /* преобразовать args: Python->C */
        return NULL;                            /* NULL возбуждает исключительную ситуацию */
    if (top == MAXSTACK)            /* python устанавливает исключение ошибки аргументов */
        onError("stack overflow")               /* iff maxstack < maxchars */
    if (len + strlen(pstr) + 1 >= MAXCHARS)
        onError("string-space overflow")
    else {
        strcpy(strings + len, pstr);            /* запись в память для строк */
        stack[top++] = &(strings[len]);         /* протолкнуть начальный адрес */
        len += (strlen(pstr) + 1);              /* новый размер памяти для строк */
        Py_INCREF(Py_None);                     /* вызов 'процедуры' *.
        return Py_None;                         /* None: ошибок нет*/
    }
}

static PyObject *
stack_pop(PyObject *self, PyObject *args)
{                                               /* pop без аргументов */
    PyObject *pstr;
    if (!PyArg_ParseTuple(args, ""))            /* проверка отсутствия аргументов */
        return NULL;
    if (top == 0)
        onError("stack underflow")              /* возврат NULL = возбудить */
    else {
        pstr = Py_BuildValue("s", stack[--top]); /* преобразовать результат: C->Py */
        len -= (strlen(stack[top]) + 1);
        return pstr;                            /* возврат новой строки python */
    }                                           /* уже pstr ref-count++ */
}

static PyObject *
stack_top(PyObject *self, PyObject *args)       /* почти как item(-1) */
{                                               /* но другие ошибки */
    PyObject *result = stack_pop(self, args);   /* получить строку с вершины */
    if (result != NULL)
        len += (strlen(stack[top++]) + 1);      /* отменить pop */
    return result;                              /* NULL или объект строки *.
}

static PyObject *
stack_empty(PyObject *self, PyObject *args)     /* нет аргументов: '()' */
{
    if (!PyArg_ParseTuple(args, ""))            /* или PyArg_NoArgs */
        return NULL;
    return Py_BuildValue("i", top == 0);        /* boolean: python int */
}
```

```
static PyObject *
stack_member(PyObject *self, PyObject *args)
{
    int i;
    char *pstr;
    if (!PyArg_ParseTuple(args, "s", &pstr))
        return NULL;
    for (i = 0; i < top; i++)                    /* найти аргумент в стеке */
        if (strcmp(pstr, stack[i]) == 0)
            return PyInt_FromLong(1);            /* возврат python int */
    return PyInt_FromLong(0);                    /* то же, что Py_BuildValue("i" */
}

static PyObject *
stack_item(PyObject *self, PyObject *args)       /* возврат Python string или NULL */
{                                                /* входные данные = (index): Python int */
    int index;
    if (!PyArg_ParseTuple(args, "i", &index))    /* преобразовать аргументы в C */
        return NULL;                             /* неверен тип или число аргументов? */
    if (index < 0)
        index = top + index;                     /* отрицательный: смещение от конца */
    if (index < 0 || index >= top)
        onError("index out-of-bounds")           /* возврат NULL = 'возбудить' */
    else
        return Py_BuildValue("s", stack[index]); /* преобразовать результат в Python */
}                                                /* не нужно INCREF для нового объекта */

static PyObject *
stack_len(PyObject *self, PyObject *args)        /* возврат Python int или NULL */
{                                                /* нет входных данных */
    if (!PyArg_ParseTuple(args, ""))
        return NULL;
    return PyInt_FromLong(top);                  /* заключить в объект python */
}

static PyObject *
stack_dump(PyObject *self, PyObject *args)       /* не "print": зарезервированное слово*/
{
    int i;
    if (!PyArg_ParseTuple(args, ""))
        return NULL;
    printf("[Stack:\n");
    for (i=top-1; i >= 0; i--)                   /* форматированный вывод */
        printf("%d: '%s'\n", i, stack[i]);
    printf("]\n");
    Py_INCREF(Py_None);
    return Py_None;
}

/*******************************************************************************
* ТАБЛИЦА РЕГИСТРАЦИИ МЕТОДОВ: ИМЯ -> УКАЗАТЕЛЬ ФУНКЦИИ
*******************************************************************************/

static struct PyMethodDef stack_methods[] = {
 {"push",       stack_push,     1},              /* имя, адрес */
 {"pop",        stack_pop,      1},              /* '1'=всегда кортеж аргументов */
 {"top",        stack_top,      1},
 {"empty",      stack_empty,    1},
 {"member",     stack_member,   1},
```

```
 {"item",       stack_item,     1},
 {"len",        stack_len,      1},
 {"dump",       stack_dump,     1},
 {NULL,         NULL}                           /* конец, для initmodule */
};

/*********************************************************************
* ФУНКЦИЯ ИНИЦИАЛИЗАЦИИ (ВО ВРЕМЯ ИМПОРТА)
*********************************************************************/

void
initstackmod()
{
    PyObject *m, *d;

    /* создать модуль и добавить функции */
    m = Py_InitModule("stackmod", stack_methods);        /* ловушка для регистрации */

    /* добавить в модуль символические константы */
    d = PyModule_GetDict(m);
    ErrorObject = Py_BuildValue("s", "stackmod.error");  /* экспорт исключений */
    PyDict_SetItemString(d, "error", ErrorObject);       /* при необходимости добавить еще */

    /* check for errors */
    if (PyErr_Occurred())
        Py_FatalError("can't initialize module stackmod");
}
```

Этот файл расширения на C компилируется и компонуется с интерпретатором статически или динамически, так же как в предыдущих примерах. Файл *makefile.stack* с CD управляет сборкой с помощью следующего правила:

```
stackmod.so: stackmod.c
    gcc stackmod.c -g -I$(PY)/Include -I$(PY) -fpic -shared -o stackmod.so
```

Смысл реализации такого стека в модуле расширения на C (помимо показа вызовов API для учебника Python) заключен в *оптимизации*: теоретически этот код должен предоставлять интерфейс, сходный с интерфейсом модуля стека Python, который мы написали ранее, но выполняться значительно быстрее благодаря тому, что написан на C. Интерфейс примерно тот же, хотя при переходе на C мы частично пожертвовали гибкостью Python – возникли ограничения на размер и типы помещаемых на стек объектов:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Stacks]$ python
>>> import stackmod                                  # загрузка модуля C
>>> stackmod.push('new')                             # вызов функций C
>>> stackmod.dump()                                  # формат dump другой
[Stack:
0: 'new'
]
>>> for c in "SPAM": stackmod.push(c)
...
>>> stackmod.dump()
[Stack:
4: 'M'
3: 'A'
2: 'P'
1: 'S'
0: 'new'
]
```

```
>>> stackmod.len(), stackmod.top()
(5, 'M')
>>> x = stackmod.pop()
>>> x
'M'
>>> stackmod.dump()
[Stack:
3: 'A'
2: 'P'
1: 'S'
0: 'new'
]
>>> stackmod.push(99)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: argument 1: expected string, int found
```

Некоторые ограничения на тип и размер в стеке C можно снять, изменив код C (и в итоге создать конструкцию, по виду и функциям почти совпадающую со встроенным списком Python). Прежде чем проверять этот стек на скорость, посмотрим, нельзя ли также оптимизировать наши *классы* стека с помощью *типа* C.

Но и этого тоже не делайте – SWIG

Можно писать такие модули вручную, но делать это не обязательно. Как было показано выше, если написать функции модуля стека без всякого представления об интеграции с Python, их можно автоматически интегрировать с Python, пропустив сигнатуры типов через SWIG. Я не стал здесь так писать эти функции, потому что мне нужно также показать лежащий в основе API расширений C для Python. Но если бы мне потребовалось написать на C стек строк для Python в любом другом контексте, я бы сделал это с помощью SWIG.

Тип стека строк: расширение на C

Чтобы реализовать на C объекты с множественными экземплярами, нужно вместо модуля написать расширение *типа* на C. Подобно классам Python типы C создают объекты с множественными экземплярами и могут перегружать (то есть перехватывать и реализовывать) операторы выражений и действия с типами Python. Однако, в отличие от классов, типы сами не поддерживают наследование атрибутов – атрибуты загружаются из плоской таблицы имен, а не дерева объектов пространства имен. Это имеет смысл, если понять, что встроенные типы Python суть просто готовые типы, закодированные как расширения на C; например, при вызове метода списка `append` никакого наследования не происходит. Добавить к типам наследование можно, написав классы «оболочки», но это ручной процесс (подробнее об этом ниже).

Однако одним из самых крупных недостатков типов является их размер – чтобы реализовать в достаточной мере оснащенный тип C, нужно написать на C много малоприятного кода и поместить в таблицы описателей типа указатели для связи с обработчиками операций. В действительности типы, расширенные на C, настолько сложны, что я хочу здесь сократить некоторые детали. Чтобы дать представление об их общей структуре, в примере 19.15 представлена реализация типа стека строк на C, из которой убраны тела всех функций. Полная реализация есть в этом файле на прилагаемом CD.

Этот тип C реализует примерно тот же интерфейс, что и классы стеков, с которыми мы встречались в главе 17, но накладывает некоторые ограничения на сам стек и не

поддерживает специализацию с помощью подклассов (это тип, а не класс). В удаленном коде используются те же алгоритмы, что и в модуле C примера 19.14, но они действуют над передаваемым им объектом `self`, который теперь ссылается на конкретный обрабатываемый объект экземпляра типа, как это делает первый аргумент в методах классов. В типах `self` служит указателем на расположенную в памяти конструкцию C `struct`, представляющую объект экземпляра типа.

Пример 19.15. PP2E\Integrate\Extend\Stacks\stacktyp.c

```
/*****************************************************************************
 * stacktyp.c: тип данных стека строк символов; тип расширения C для использования в программах
 * Python; клиенты модуля stacktype могут создавать множественные стеки; сходен с stackmod,
 * но 'self' является экземпляром, и можно перегружать операторы последовательностей;
 *****************************************************************************/

#include "Python.h"

static PyObject *ErrorObject;      /* локальное исключение */
#define onError(message) \
       { PyErr_SetString(ErrorObject, message); return NULL; }

/*****************************************************************************
 * ИНФОРМАЦИЯ ТИПА СТЕКА
 *****************************************************************************/

#define MAXCHARS 2048
#define MAXSTACK MAXCHARS

typedef struct {                /* формат объекта экземпляра стека */
    PyObject_HEAD               /* заголовок python: ref-count + &typeobject */
    int top, len;
    char *stack[MAXSTACK];      /* данные состояния каждого экземпляра */
    char strings[MAXCHARS];     /* то же, что stackmod, но несколько экземпляров*/
} stackobject;

/*****************************************************************************
 * МЕТОДЫ ЭКЗЕМПЛЯРА
 *****************************************************************************/

static PyObject *               /* при "instance.push(arg)" */
stack_push(self, args)          /* 'self' есть объект экземпляра стека */
    stackobject *self;          /* 'args' - аргументы, переданные в метод self.push */
    PyObject    *args;
{    ...
}
static PyObject *
stack_pop(self, args)
    stackobject *self;
    PyObject    *args;          /* при "instance.pop()" */
{    ...
}
static PyObject *
stack_top(self, args)
    stackobject *self;
    PyObject    *args;
{    ...
}
static PyObject *
stack_empty(self, args)
    stackobject *self;
```

```
    PyObject    *args;
{    ...
}
static struct PyMethodDef stack_methods[] = {  /* методы экземпляра */
 {"push",       stack_push,     1},            /* таблица имен/адресов */
 {"pop",        stack_pop,      1},            /* как append,sort списка */
 {"top",        stack_top,      1},
 {"empty",      stack_empty,    1},            /* дополнительные операции сверх операторов */
 {NULL,         NULL}                          /* конец, для getattr */
};

/*******************************************************************************
 * БАЗОВЫЕ ОПЕРАЦИИ ТИПА
 *******************************************************************************/

static stackobject *               /* при "x = stacktype.Stack()" */
newstackobject()                   /* функция-конструктор экземпляра */
{   ...                            /* не получают на входе `args` */
}
static void                        /* функция-деструктор экземпляра */
stack_dealloc(self)                /* при достижении счетчиком ссылок нуля */
    stackobject *self;
{   ...                            /* уборка */
}
static int
stack_print(self, fp, flags)
    stackobject *self;
    FILE *fp;
    int flags;                     /* вывод self в файл */
{   ...
}
static PyObject *
stack_getattr(self, name)          /* при обращении "instance.attr" */
    stackobject *self;             /* создать связанный метод или член */
    char *name;
{   ...
}
static int
stack_compare(v, w)                /* для всех сравнений */
    stackobject *v, *w;
{   ...
}

/*******************************************************************************
 * ОПЕРАЦИИ ТИПА ПОСЛЕДОВАТЕЛЬНОСТИ
 *******************************************************************************/

static int
stack_length(self)
    stackobject *self;               /* вызывается при "len(экземпляр)" */
{   ...
}
static PyObject *
stack_concat(self, other)
    stackobject *self;               /* при "экземпляр + other" */
    PyObject    *other;              /* `self` - экземпляр */
{   ...
}
static PyObject *
```

```
stack_repeat(self, n)                /* при "instance * N" */
    stackobject *self;               /* новый стек = повторить self n раз */
    int n;
{   ...
}
static PyObject *
stack_item(self, index)              /* при "instance[offset]", "in/for" */
    stackobject *self;               /* возврат i-го элемента self */
    int index;                       /* отрицательные индексы перестраиваются */
{   ...
}
static PyObject *
stack_slice(self, ilow, ihigh)
    stackobject *self;               /* при "instance[ilow:ihigh]" */
    int ilow, ihigh;                 /* отрицательные-перестраиваются, не увеличиваются */
{   ...
}

/*****************************************************************************
 * ОПИСАТЕЛИ ТИПА
 *****************************************************************************/

static PySequenceMethods stack_as_sequence = {  /* дополнение последовательности */
      (inquiry)       stack_length,             /* sq_length    "len(x)"   */
      (binaryfunc)    stack_concat,             /* sq_concat    "x + y"    */
      (intargfunc)    stack_repeat,             /* sq_repeat    "x * n"    */
      (intargfunc)    stack_item,               /* sq_item      "x[i], in" */
      (intintargfunc) stack_slice,              /* sq_slice     "x[i:j]"   */
      (intobjargproc)     0,                    /* sq_ass_item  "x[i] = v" */
      (intintobjargproc)  0,                    /* sq_ass_slice "x[i:j]=v" */
};

static PyTypeObject Stacktype = {       /* главный описатель типа python */
  /* заголовок типа */                  /* используется всеми экземплярами */
      PyObject_HEAD_INIT(&PyType_Type)
      0,                                /* ob_size */
      "stack",                          /* tp_name */
      sizeof(stackobject),              /* tp_basicsize */
      0,                                /* tp_itemsize */

  /* стандартные методы */
      (destructor)  stack_dealloc,      /* tp_dealloc  ref-count==0  */
      (printfunc)   stack_print,        /* tp_print    "print x"     */
      (getattrfunc) stack_getattr,      /* tp_getattr  "x.attr"      */
      (setattrfunc) 0,                  /* tp_setattr  "x.attr=v"    */
      (cmpfunc)     stack_compare,      /* tp_compare  "x > y"       */
      (reprfunc)    0,                  /* tp_repr     `x`, print x  */

  /* категории типа */
      0,                                /* tp_as_number   +,-,*,/,%,&,>>,...*/
      &stack_as_sequence,               /* tp_as_sequence +,[i],[i:j],len, ...*/
      0,                                /* tp_as_mapping  [key], len, ...*/

  /* другие методы */
      (hashfunc)     0,                 /* tp_hash    "dict[x]" */
      (ternaryfunc)  0,                 /* tp_call    "x()"     */
      (reprfunc)     0,                 /* tp_str     "str(x)"  */

};  /* плюс остальные: см. Include/object.h */
```

```
/*****************************************************************************
 * ЛОГИКА МОДУЛЯ
 *****************************************************************************/

static PyObject *
stacktype_new(self, args)                       /* при "x = stacktype.Stack()" */
    PyObject *self;                             /* self не используется */
    PyObject *args;                             /* аргументы конструктора */
{
    if (!PyArg_ParseTuple(args, ""))            /* функция метода модуля */
        return NULL;
    return (PyObject *)newstackobject();        /* создать новый объект экземпляра типа */
}                                               /* привязка модуля к типу... */

static struct PyMethodDef stacktype_methods[] = {
    {"Stack",  stacktype_new,  1},              /* одна функция: создать стек */
    {NULL,     NULL}                            /* маркер конца, для initmodule */
};

void
initstacktype()                                 /* при первом "import stacktype" */
{
    PyObject *m, *d;
    m = Py_InitModule("stacktype", stacktype_methods);   /* создать модуль, */
    d = PyModule_GetDict(m);                             /* функция 'Stack' */
    ErrorObject = Py_BuildValue("s", "stacktype.error");
    PyDict_SetItemString(d, "error", ErrorObject);       /* экспорт исключения */
    if (PyErr_Occurred())
        Py_FatalError("can't initialize module stacktype");
}
```

Анатомия типа, расширенного на C

Хотя большая часть файла *stacktyp.c* отсутствует, оставшейся достаточно для иллюстрации глобальной структуры, присущей реализациям типов на C:

Структура экземпляра

Файл начинается с определения конструкции C struct с именем `stackobject`, в которой будет храниться информация по каждому экземпляру – каждый создаваемый объект экземпляра получает через `malloc` свою копию struct. Она выполняет ту же функцию, что и словари атрибутов экземпляров класса, и содержит данные, сохраненные модулем стека на C в глобальных переменных.

Методы экземпляра

После этого следует группа методов экземпляра, как и в модуле; они реализуют вызовы таких методов, как `push` и `pop`. Но в данном случае функции методов обрабатывают предполагаемый объект экземпляра, переданный в аргументе `self`. Это близко по духу к методам класса. При обращении к методам экземпляра происходит их поиск в таблице регистрации (см. листинг примера 19.15).

Базовые операции типов

После этого в файле определены функции, осуществляющие базовые операции, имеющиеся во всех типах: создание, вывод, квалификация и т. д. Сигнатуры типов этих функций более специфичны, чем у обработчиков методов экземпляров. Обработчик создания объекта размещает в памяти `struct` нового стека и инициализирует его поля заголовка: счетчик ссылок устанавливается в 1, а указатель объекта типа устанавливается равным дескриптору типа `Stacktype`, который находится далее в файле.

Операции последовательностей

Далее идут функции выполнения операций типа последовательности. К стекам применимо большинство операторов последовательностей: `len`, `+`, `*` и `[i]`. Аналогично методу класса `__getitem__`, функция `stack_item` обрабатывает доступ по индексу, а также проверку вхождения `in` и циклы `for`. Две последние операции осуществляются обращением к объекту по индексу, пока Python не перехватит исключительную ситуацию `IndexError`.

Описатели типа

Таблицы описателей типа (в действительности `struct`), находящиеся в конце файла, составляют главный вопрос для типов – с помощью этих таблиц Python переправляет операцию, выполняемую над объектом экземпляра соответствующей обрабатывающей функции C в этом файле. На самом деле через эти таблицы маршрутизируется все; даже поиск атрибутов методов начинается с выполнения функции C `stack_getattr`, приведенной в этой таблице (которая, в свою очередь, ищет имя атрибута в таблице имен/адресов функций). В главной таблице `Stacktype` есть ссылка на дополнительную таблицу `stack_as_sequence`, в которой зарегистрированы обработчики операций последовательностей; для типов в таких таблицах могут регистрироваться обработчики групп операций соответствий, чисел и последовательностей. Примеры для чисел и соответствий можно найти в исходном коде объектов Python чисел и словарей; здесь они аналогичны типу последовательности, но таблицы операций отличаются.[1]

Модуль конструктора

Помимо определения типа C в конце этого файла создается простой *модуль* C, экспортирующий функцию конструктора `stacktype.Stack`, которую вызывают сценарии Python для генерации новых объектов экземпляров стека. Функция инициализации для этого модуля представляет единственное имя C в этом файле, которое не определено как `static` (локальное для файла); доступ ко всему остальному осуществляется через указатели – от экземпляра до описателя типа и функции-обработчика на C .

Опять-таки, полную реализацию типа стека C можно найти на прилагаемом к книге CD. Но чтобы дать вам представление о том, как выглядят методы типа C, приведем код функции `pop` типа C; сравните его с функцией `pop` модуля C и посмотрите, как аргумент `self` используется для доступа к данным экземпляров в типах:

```
static PyObject *
stack_pop(self, args)
    stackobject *self;
    PyObject *args;                          /* при "instance.pop()" */
{
    PyObject *pstr;
    if (!PyArg_ParseTuple(args, ""))         /* проверка отсутствия аргументов*/
        return NULL;
    if (self->top == 0)
        onError("stack underflow")           /* возврат NULL = возбудить */
    else {
```

[1] Учтите, что структура описателей типов, как и большинство средств C API, со временем может измениться, и всегда нужно проверять самый свежий список полей в файле *Include/object.h* дистрибутива Python. В новых версиях Python может также потребоваться перекомпиляция типов, написанных для работы с предыдущими версиями, чтобы восстановить соответствие с изменениями в описателях. Как всегда, дополнительную информацию и примеры можно найти в руководствах Python по расширениям и полном исходном коде.

```
        pstr = Py_BuildValue("s", self->stack[--self->top]);
        self->len -= (strlen(self->stack[self->top]) + 1);
        return pstr;
    }
}
```

Компиляция и выполнение

Этот файл расширения на C компилируется и компонуется динамически или статически, как предшествующие примеры; файл *makefile.stack* на CD выполняет компиляцию так:

```
stacktype.so: stacktyp.c
    gcc stacktyp.c -g -I$(PY)/Include -I$(PY) -fpic -shared -o stacktype.so
```

После компиляции можно импортировать модуль C и создавать и пользоваться экземплярами определенного в нем типа C, почти как если бы это был класс Python (но без наследования). Обычно это делается из сценария Python, но удобно проверить основы в интерактивном приглашении:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Stacks]$ python
>>> import stacktype                        # импорт модуля конструктора C
>>> x = stacktype.Stack()                   # создать объект экземпляра типа C
>>> x.push('new')                           # вызвать методы типа C
>>> x                                       # вызвать обработчик print типа C
[Stack:
0: 'new'
]

>>> x[0]                                    # вызвать обработчик доступа по индексу типа C
'new'
>>> y = stacktype.Stack()                   # создать еще один экземпляр типа
>>> for c in 'SPAM': y.push(c)              # отдельный объект стека
...
>>> y
[Stack:
3: 'M'
2: 'A'
1: 'P'
0: 'S'
]

>>> z = x + y                               # вызвать обработчик конкатенации типа C
>>> z
[Stack:
4: 'M'
3: 'A'
2: 'P'
1: 'S'
0: 'new'
]

>>> y.pop()
'M'
>>> len(z), z[0], z[-1]                    # циклы for тоже работают (доступ по индексу)
(5, 'new', 'M')
```

Хронометраж реализаций на C

Что же у нас на этот раз получилось с оптимизацией? Воскресим модуль таймера, который мы написали в примере 17.6, чтобы сравнить модуль и тип стека C с модулем стека и классами Python, которые мы написали в главе 17. Пример 19.16 вычисляет системное время, которое требуется для выполнения тестов по всем реализациям стеков в этой книге, в секундах.

Пример 19.16. PP2E\Integrate\Extend\Stacks\exttime.py

```
#!/usr/local/bin/python
# хронометраж расширений модуля и типа стека на C
# в сравнении с реализациями стека на Python в главе по объектам

from PP2E.Dstruct.Basic.timer  import test      # функция подсчета секунд
from PP2E.Dstruct.Basic import stack1           # модуль стека python
from PP2E.Dstruct.Basic import stack2           # класс стека python: +/срез
from PP2E.Dstruct.Basic import stack3           # класс стека python: кортежи
from PP2E.Dstruct.Basic import stack4           # класс стека python: append/pop
import stackmod, stacktype                      # тип расширения C, модуль

from sys import argv
rept, pushes, pops, items = 200, 200, 200, 200  # по умолчанию: 200 * (600 операций)
try:
    [rept, pushes, pops, items] = map(int, argv[1:])
except: pass
print 'reps=%d * [push=%d+pop=%d+fetch=%d]' % (rept, pushes, pops, items)

def moduleops(mod):
    for i in range(pushes): mod.push('hello')   # строки только для C
    for i in range(items):  t = mod.item(i)
    for i in range(pops):   mod.pop()

def objectops(Maker):                           # в типе нет аргументов инициализации
    x = Maker()                                 # экземпляр типа или класса
    for i in range(pushes): x.push('hello')     # строки только для C
    for i in range(items):  t = x[i]
    for i in range(pops):   x.pop()

# тестирование модулей: python/c
print "Python module:", test(rept, moduleops, stack1)
print "C ext module: ", test(rept, moduleops, stackmod), '\n'

# тестирование объектов: класс/тип
print "Python simple Stack:", test(rept, objectops, stack2.Stack)
print "Python tuple  Stack:", test(rept, objectops, stack3.Stack)
print "Python append Stack:", test(rept, objectops, stack4.Stack)
print "C ext type Stack:   ", test(rept, objectops, stacktype.Stack)
```

При выполнении этого сценария под Linux получаются показанные ниже результаты. Как мы уже говорили, стек кортежей Python несколько лучше, чем `append` стека по месту в Python при типичном использовании (когда со стеком выполняются только push и pop), но медленнее при обращении по индексу. Первый тест здесь выполняет 200 повторений 200 push и pop стека или 80 000 операций со стеком (200 × 400); приводимое время выражает продолжительность теста в секундах:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Stacks]$ python exttim.py 200 200 200 0
reps=200 * [push=200+pop=200+fetch=0]
Python module: 2.09
C ext module:  0.68
```

```
Python simple Stack: 2.15
Python tuple  Stack: 0.68
Python append Stack: 1.16
C ext type Stack:    0.5

[mark@toy ~/.../PP2E/Integrate/Extend/Stacks]$ python exttim.py 100 300 300 0
reps=100 * [push=300+pop=300+fetch=0]
Python module: 1.86
C ext module:  0.52

Python simple Stack: 1.91
Python tuple  Stack: 0.51
Python append Stack: 0.87
C ext type Stack:    0.38
```

По крайней мере в отсутствие операций обращения к стеку по индексу, как в этих двух тестах (только push и pop), тип на C лишь не намного быстрее, чем лучший стек Python (кортежи). На самом деле здесь почти ничья – в этих первых двух тестах тип C оказывается только на одну десятую секунды быстрее после 200 стеков и 80 000 операций со стеком. Это не того рода различие в производительности, чтобы писать сообщение об ошибке.[1]

Модуль на C оказывается примерно втрое быстрее модуля Python, но эти результаты некорректны. Тестируемый здесь модуль Python `stack1` использует ту же медленную реализацию стека, что и «простой» стек Python (`stack2`). Если бы его переписать с использованием представления стека в виде кортежей, как в главе 17, его скорость была бы аналогичной приведенным здесь цифрам для «кортежей» и почти совпадающей со скоростью модуля C в первых двух тестах:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Stacks]$ python exttim.py 200 200 200 50
reps=200 * [push=200+pop=200+fetch=50]
Python module: 2.17
C ext module:  0.79

Python simple Stack: 2.24
Python tuple  Stack: 1.94
Python append Stack: 1.25
C ext type Stack:    0.52

[mark@toy ~/.../PP2E/Integrate/Extend/Stacks]$ python exttim.py
reps=200 * [push=200+pop=200+fetch=200]
Python module: 2.42
C ext module:  1.1

Python simple Stack: 2.54
Python tuple  Stack: 19.09
Python append Stack: 1.54
C ext type Stack:    0.63
```

Но в других схемах применения, моделируемых этими двумя тестами, побеждает тип C. Он примерно вдвое быстрее лучшего стека Python (`append`), когда в тест добавляется доступ по индексу, как проиллюстрировано двумя из предыдущих прогонов тестов, выполненных с ненулевым счетчиком выборки. Аналогично модуль на C тоже будет в этом случае вдвое быстрее лучшего кода модуля на Python.

---

[1] Интересно, что Python стал работать значительно быстрее с момента выхода первого издания этой книги по отношению к C. В то время тип C был почти втрое быстрее лучшего стека Python (кортежи) в отсутствие доступа по индексу. Сегодня они идут почти вровень. Отсюда можно сделать вывод, что переход на C стал иметь втрое меньшее значение, чем когда-то.

Иными словами, самые быстрые стеки Python так же хороши, как стеки C, если ограничиваться push и pop, но стеки C примерно вдвое быстрее, если производится доступ по индексу. Кроме того, поскольку нужно выбрать одно представление, то если доступ по индексу вообще возможен, нужно, видимо, выбрать стек Python `append`; если стеки C представляют наилучший случай, они всегда оказываются вдвое быстрее.

Конечно, измеренные временные различия настолько малы, что во многих приложениях они несущественны. Кроме того, программировать стеки C гораздо труднее, а их скорость достигается при наложении существенных функциональных ограничений; во многих случаях сравнивать их не вполне корректно. Но практика показывает, что расширения на C не только интегрируют существующие компоненты для использования в сценариях Python, но могут оптимизировать критические по времени компоненты в чистых программах Python. В некоторых ситуациях переход на C может дать еще более значительное ускорение.

С другой стороны, к расширениям на C обычно следует прибегать только в крайних случаях. Как мы уже отмечали, алгоритмы и структура данных часто оказывают на производительность программ более заметное влияние, чем язык, использованный для реализации. Тот факт, что написанные на Python стеки на основе кортежей в обычных схемах применения почти столь же быстры, как стеки C, многое говорит о важности структуры представления данных.

Классы-оболочки для типов C

В текущей реализации Python для добавления наследования к типам C нужно использовать классы. Чаще всего для поддержки индивидуальной подгонки типов применяются классы-оболочки (*wrapper*) – классы Python, главная задача которых состоит в хранении ссылки на объект типа и передачи ему всех операций. Такая оболочка вводит интерфейс класса поверх типа, благодаря чему появляется возможность создания подклассов для базового типа и расширения его, как если бы он являлся классом. Это иллюстрируется примером 19.17.

Пример 19.17. PP2E\Integrate\Extend\Stacks\oopstack.py

```
import stacktype                                # получить тип/модуль C
class Stack:
    def __init__(self, start=None):             # создать/обернуть экземпляр типа C
        self._base = start or stacktype.Stack()  # удаляется вместе с классом экземпляра
    def __getattr__(self, name):
        return getattr(self._base, name)         # методы/члены: экземпляр типа
    def __cmp__(self, other):
        return cmp(self._base, other)
    def __repr__(self):                          # 'print' отличен от repr
        print self._base,; return ''
    def __add__(self, other):                    # операторы: специальные методы
        return Stack(self._base + other._base)   # операторы – не атрибуты
    def __mul__(self, n):
        return Stack(self._base * n)             # заключить результат в новый Stack
    def __getitem__(self, i):
        return self._base[i]                     # 'item': индекс, in, for
    def __len__(self):
        return len(self._base)
```

Этот класс оболочки можно использовать так же, как тип C, потому что он передает все операции экземпляру типа, хранящемуся в `self._base` экземпляра класса:

```
[mark@toy ~/.../PP2E/Integrate/Extend/Stacks]$ python
```

```
>>> import oopstack
>>> x = oopstack.Stack()
>>> y = oopstack.Stack()
>>> x.push('class')
>>> for c in "SPAM": y.push(c)
...
>>> x
[Stack:
0: 'class'
]

>>> y[2]
'A'
>>> z = x + y
>>> for s in z: print s,
...
class S P A M

>>> z.__methods__, z.__members__, z.pop()
(['empty', 'pop', 'push', 'top'], ['len'], 'M')
>>> type(z), type(z._base)
(<type 'instance'>, <type 'stack'>)
```

Целью кодирования такой оболочки является улучшение поддержки расширений в Python. Подклассы в действительности являются подклассами класса оболочки, но поскольку оболочка является просто тонким интерфейсом к типу, она почти является подклассом самого типа, как в примере 19.18.

Пример 19.18. PP2E\Integrate\Extend\Stacks\substack.py

```
from oopstack import Stack          # получить класс 'stub' (оболочка типа C)

class Substack(Stack):
    def __init__(self, start=[]):   # расширить операцию 'new'
        Stack.__init__(self)        # инициализация стека любой последовательностью
        for str in start:           # start может быть другим стеком
            self.push(str)
    def morestuff(self):            # добавить новый метод
        print 'more stack stuff'
    def __getitem__(self, i):       # расширить 'item' для трассировки обращений
        print 'accessing cell', i
        return Stack.__getitem__(self, i)
```

Этот подкласс расширяет тип (оболочку), поддерживая установку начального значения на этапе создания, выводя сообщения трассировки при доступе по индексу и вводя совершенно новый метод `morestuff`. Этот подкласс ограничен (например, результатом + является `Stack`, а не `Substack`), но подтверждает идею – оболочки позволяют воспользоваться техникой наследования и композиции, изученной в этой книге, для новых типов, написанных на C:

```
>>> import substack
>>> a = substack.Substack(x + y)
>>> a
[Stack:
4: 'M'
3: 'A'
2: 'P'
1: 'S'
0: 'class'
]
```

```
>>> a[3]
accessing cell 3
'A'
>>> a.morestuff()
more stack stuff
>>> b = substack.Substack("C" + "++")
>>> b.pop(), b.pop()
('+', '+')
>>> c = b + substack.Substack(['-', '-'])
>>> for s in c: print s,
...
C - -
```

Но этого тоже ненужно делать – пользуйтесь SWIG

Можно вручную кодировать типы C подобным образом, но делать это необязательно. Так как SWIG умеет генерировать связующий код для классов C++, можно *автоматически* генерировать весь код расширения C и класса оболочки для интеграции такого объекта стека, выполнив SWIG над соответствующим объявлением класса. В следующем разделе показано, как это делается.

Создание оболочек классов C++ с помощью SWIG

Более ловким трюком, осуществляемым SWIG, является генерация оболочки класса – по объявлению класса C++ и с помощью специальных параметров командной строки SWIG генерирует:

- Написанный на C++ модуль расширения Python с функциями доступа, взаимодействующими с методами и членами класса C++
- Написанный на Python класс оболочки (называемый в SWIG «теневым» классом), взаимодействующий с модулем функций доступа к классу C++

Как и прежде, нужно просто запустить SWIG в make-файле, чтобы просмотреть объявление класса C++ и скомпилировать вывод. В итоге при импорте теневого класса в сценариях Python можно использовать классы C++, как если бы они были написаны на Python. Программы Python не только могут создавать и использовать экземпляры классов C++, но и выполнять их индивидуальную подгонку путем создания подклассов сгенерированного теневого класса.

Маленький (но не слишком) класс C++

Чтобы увидеть, как все это работает, нам нужен класс C++. Для иллюстрации напишем простой класс, который будет использоваться в сценариях Python.[1] В следующих файлах C++ определен класс `Number` с тремя методами (`add`, `sub`, `display`), переменной класса (`data`), конструктором и деструктором. В примере 19.19 показан файл заголовка.

Пример 19.19. PP2E\Integrate\Extend\Swig\Shadow\number.h

```
class Number
{
```

[1] Для более прямого сравнения можно перевести тип стека также в класс C++, но тогда получится гораздо больше кода C++, чем хотелось бы показывать в этой книге по Python. Кроме того, при этом переводе пришлось бы пожертвовать возможностями перегрузки операторов типа (SWIG в данное время не поддерживает перегрузку операторов C++).

```
public:
    Number(int start);
    ~Number();
    void add(int value);
    void sub(int value);
    void display();
    int data;
};
```

В примере 19.20 показана реализация класса C++; при вызове каждого метода выводится сообщение трассировки операций класса.

Пример 19.20. PP2E\Integrate\Extend\Swig\Shadow\number.cxx

```
#include "number.h"
#include "iostream.h"
// #include "stdio.h"

Number::Number(int start) {
   data = start;
   cout << "Number: " << data << endl;    // работают cout и printf
   // printf("Number: %d\n", data);       // print python выводит в stdout
}

Number::~Number() {
   cout << "~Number: " << data << endl;
}

void Number::add(int value) {
   data += value;
   cout << "add " << value << endl;
}

void Number::sub(int value) {
   data -= value;
   cout << "sub " << value << endl;
}

void Number::display() {
   cout << "Number = " << data << endl;
}
```

Чтобы сравнить языки, посмотрим, как можно использовать этот класс в программе C++; в примере 19.21 создается объект `Number`, вызываются его методы, напрямую загружается и устанавливается его атрибут данных (в C++ делается различие между «данными» и «методами», в то время как в Python те и другие обычно называют «атрибутами»).

Пример 19.21. PP2E\Integrate\Extend\Swig\Shadow\main.cxx

```
#include "iostream.h"
#include "number.h"

main()
{
    Number *num;
    num = new Number(1);            // создать экземпляр класса C++
    num->add(4);                    // вызвать его методы
    num->display();
    num->sub(2);
    num->display();
```

```
    num->data = 99;                 // установить данные C++
    cout << num->data << endl;      // получить данные C++
    num->display();
    delete num;
}
```

Для компиляции и запуска этого кода в Linux можно использовать компилятор C++ командной строки `g++`. Если вы работаете не в Linux, вам придется импровизировать (между компиляторами C++ слишком много различий, чтобы их здесь перечислять).

```
[mark@toy ~/.../PP2E/Integrate/Extend/Swig/Shadow]$ g++ main.cxx number.cxx
[mark@toy ~/.../PP2E/Integrate/Extend/Swig/Shadow]$ a.out
Number: 1
add 4
Number = 5
sub 2
Number = 3
99
Number = 99
~Number: 99
```

Создание оболочки для класса на C++ с помощью SWIG

Вернемся к Python. Чтобы использовать класс C++ `Number` в сценариях Python, нужно написать или сгенерировать слой связующей логики между этими двумя языками, как в предыдущих примерах. Для автоматической генерации этого слоя нужно написать входной файл SWIG, вроде приведенного в примере 19.22.

Пример 19.22. PP2E\Integrate\Extend\Swig\Shadow\number.i

```
/*****************************************************************
 * Файл описания модуля Swig для создания оболочки класса C++.
 * Генерация по команде "swig -python -shadow number.i".
 * Модуль C создается в файле number_wrap.c;
 * здесь модуль 'number' указывает на теневой класс number.py.
 *****************************************************************/

%module number

%{
#include "number.h"
%}

%include number.h
```

Этот интерфейс просто указывает SWIG, что он должен прочесть данные с сигнатурой типа класса C++ из включаемого заголовочного файла *number.h*. На этот раз SWIG создает по объявлению класса три файла и два разных модуля Python:

- *number_wrap.doc*, простой файл описания функции оболочки
- *number_wrap.c*, модуль расширения C++ с функциями доступа к классу
- *number.py*, теневой класс Python, служащий оболочкой функций доступа

В примере 19.23 приведен make-файл для Linux, который объединяет созданный модуль кода оболочки C++ с файлом реализации класса C++ и создает *numberc.so*, динамически загружаемый модуль расширения, который при импортировании в сценарии Python должен находиться в пути поиска модулей Python.

Пример 19.23. PP2E\Integrate\Extend\Swig\Shadow\makefile.number-swig

```
################################################################################
# Применение SWIG для интеграции класса C++ number.h с программами Python.
# Замечание: имя "numberc.so" важно, потому что теневой класс импортирует numberc.
################################################################################

# если вы не выполнили make install
SWIG = ../myswig
PY   = $(MYPY)

all: numberc.so number.py

# оболочка + действительный класс
numberc.so: number_wrap.o number.o
    g++ -shared number_wrap.o number.o -o numberc.so

# сгенерированный модуль оболочки класса
number_wrap.o: number_wrap.c number.h
    g++ number_wrap.c -c -g -I$(PY)/Include -I$(PY)

number_wrap.c: number.i
    $(SWIG) -c++ -python -shadow number.i

number.py: number.i
    $(SWIG) -c++ -python -shadow number.i

# код класса C++, заключаемого в оболочку
number.o: number.cxx number.h
    g++ -c -g number.cxx

cxxtest:
    g++ main.cxx number.cxx

clean:
    rm -f *.pyc *.o *.so core a.out
force:
    rm -f *.pyc *.o *.so core a.out number.py number_wrap.c number_wrap.doc
```

Как обычно, выполните этот make-файл, чтобы сгенерировать и откомпилировать необходимый связующий код в виде модуля расширения, который может импортироваться программами Python:

```
[mark@toy ~/...../Integrate/Extend/Swig/Shadow]$ make -f makefile.number-swig
Generating wrappers for Python
g++ number_wrap.c -c -g -I/...
g++ -c -g number.cxx
g++ -shared number_wrap.o number.o -o numberc.so
```

Чтобы несколько демистифицировать работу SWIG, ниже приведена часть сгенерированного модуля C++ с функциями доступа *number_wrap.c*. Полный файл исходного текста можно найти на прилагаемом к книге CD (или создать самостоятельно). Обратите внимание, что в этом файле определен простой модуль расширения на C с функциями, которые обычно предполагают передачу им указателя на объект C++ (то есть указатель «this» на языке C++). Эта структура несколько отличается от использованной в примере 19.17, где оболочкой *типа* C служил класс Python, но конечный результат аналогичен:

...реализации функций _wrap используют синтаксис C++...

```
#define new_Number(_swigarg0) (new Number(_swigarg0))
static PyObject *_wrap_new_Number(PyObject *self, PyObject *args) {
  ...тело удалено...
```

```
}

#define Number_add(_swigobj,_swigarg0)  (_swigobj->add(_swigarg0))
static PyObject *_wrap_Number_add(PyObject *self, PyObject *args) {
  ...тело удалено...
}

#define Number_data_get(_swigobj) ((int ) _swigobj->data)
static PyObject *_wrap_Number_data_get(PyObject *self, PyObject *args) {
  ...тело удалено...
}

static PyMethodDef numbercMethods[] = {
     { "Number_data_get", _wrap_Number_data_get, 1 },
     { "Number_data_set", _wrap_Number_data_set, 1 },
     { "Number_display", _wrap_Number_display, 1 },
     { "Number_sub", _wrap_Number_sub, 1 },
     { "Number_add", _wrap_Number_add, 1 },
     { "delete_Number", _wrap_delete_Number, 1 },
     { "new_Number", _wrap_new_Number, 1 },
     { NULL, NULL }
};

SWIGEXPORT(void,initnumberc)() {
     PyObject *m, *d;
     SWIG_globals = SWIG_newvarlink();
     m = Py_InitModule("numberc", numbercMethods);
     d = PyModule_GetDict(m);
```

Поверх модуля функций доступа SWIG генерирует *number.py*, следующий теневой класс, импортируемый сценариями Python в качестве действительного интерфейса к классу. Этот код немного сложнее, чем класс оболочки, который мы видели в предыдущем разделе, потому что он управляет владением объектами, из-за чего по-разному работает с новыми и существующими объектами. Важно отметить, что это чистый класс Python, который сохраняет указатель C++ «this» ассоциированного объекта C++ и передает управление функциям доступа в сгенерированном модуле расширения на C++:

```
import numberc
class NumberPtr :
    def __init__(self,this):
        self.this = this
        self.thisown = 0
    def __del__(self):
        if self.thisown == 1 :
            numberc.delete_Number(self.this)
    def add(self,arg0):
        val = numberc.Number_add(self.this,arg0)
        return val
    def sub(self,arg0):
        val = numberc.Number_sub(self.this,arg0)
        return val
    def display(self):
        val = numberc.Number_display(self.this)
        return val
    def __setattr__(self,name,value):
        if name == "data" :
            numberc.Number_data_set(self.this,value)
            return
        self.__dict__[name] = value
```

```
    def __getattr__(self,name):
        if name == "data" :
            return numberc.Number_data_get(self.this)
        raise AttributeError,name
    def __repr__(self):
        return "<C Number instance>"
class Number(NumberPtr):
    def __init__(self,arg0) :
        self.this = numberc.new_Number(arg0)
        self.thisown = 1
```

Тонкий момент: сгенерированный файл модуля C++ носит имя *number_wrap.c*, но в его функции инициализации задается имя модуля Python `numberc`, которое служит также именем, импортируемым теневым классом. Импорт действует потому, что комбинация модуля связующего кода и файла библиотеки C++ компонуется в файл *numberc.so*, чтобы имена импортируемого файла модуля и функции инициализации совпадали. При использовании теневых классов и динамического связывания имя скомпилированного объектного файла обычно должно быть именем модуля, заданным в файле *.i*, с добавлением «c». В целом для входного файла *interface.i*:

```
%module interface
...объявления...
```

SWIG генерирует файл связующего кода *interface_wrap.c*, который должен быть скомпилирован в файл *interfacec.so*, динамически загружаемый при импорте:

```
swig -python -shadow interface.i
g++ -c interface.c interface_wrap.c ...еще...
g++ -shared interface.o interface_wrap.o -o interfacec.so
```

Имя модуля `interface` зарезервировано для генерируемого модуля теневого класса *interface.py*. Не забывайте, что данная структура реализации может измениться по желанию создателя SWIG, но предоставляемый им интерфейс должен сохраниться – класс Python, олицетворяющий класс C++, атрибут в атрибут.[1]

Использование класса C++ в Python

Когда связующий код сгенерирован и откомпилирован, сценарии Python могут обращаться к классу C++, как если бы он был написан на Python. Пример 19.24 повторяет тесты классов файла *main.cxx*; здесь, однако, класс C++ используется из языка программирования Python.

Пример 19.24. PP2E\Integrate\Extend\Swig\Shadow\main.py

```
from number import Number       # использование класса C++ в Python (теневой класс)
                                # выполняет те же тесты, что в файле C++ main.cxx
num = Number(1)                 # создать объект класса C++ в Python
num.add(4)                      # вызвать его методы из Python
num.display()                   # num сохраняет указатель C++ 'this'
num.sub(2)
num.display()
```

[1] Пока я это писал, Гвидо несколько раз предположил, что будущая версия Python более тесно соединит понятия классов Python и типов C и даже будет переписана на C++, чтобы облегчить интеграцию с C++ в целом. Если это произойдет, то, возможно, SWIG станет использовать типы C для оболочек классов C++ вместо функций доступа плюс класс Python, как сейчас. Но это не обязательно. Следите на *http://www.swig.org* за последними событиями, не отраженными в этой книге.

```
num.data = 99                    # установка члена данных C++, генерированным __setattr__
print num.data                   # получение члена данных C++, генерированным __getattr__
num.display()
del num                          # автоматически выполняет деструктор C++
```

Так как класс C++ и его оболочки автоматически загружаются при импорте теневым классом `number`, этот сценарий выполняется, как всякий другой:

```
[mark@toy ~/...../Integrate/Extend/Swig/Shadow]$ python main.py
Number: 1
add 4
Number = 5
sub 2
Number = 3
99
Number = 99
~Number: 99
```

Этот вывод производится в основном методами класса C++ и совпадает с результатами *main.cxx* из примера 19.21. Если вы действительно хотите использовать сгенерированный модуль функций доступа, это можно сделать, как в примере 19.25.

Пример 19.25. PP2E\Integrate\Extend\Swig\Shadow\main_low.py

```
from numberc import *            # тот же тест, что в main.cxx
                                 # использовать интерфейс низкого уровня функций доступа C
num = new_Number(1)
Number_add(num, 4)               # явная передача указателя C++ 'this'
Number_display(num)              # использовать функции доступа в модуле C
Number_sub(num, 2)
Number_display(num)

Number_data_set(num, 99)
print Number_data_get(num)
Number_display(num)
delete_Number(num)
```

Этот сценарий порождает такой же вывод, как *main.py*, но преимуществ перехода к функциям от теневого класса здесь не видно. При использовании теневого класса одновременно получаются основанный на объектах интерфейс к C++ и доступный для настроек объект Python. Скажем, модуль Python, показанный в примере 19.26, расширяет класс C++, добавляя методу C++ `add` команду print и определяя новый метод `mul`. Так как теневой класс представляет собой чистый Python, это действует естественным образом.

Пример 19.26. PP2E\Integrate\Extend\Swig\Shadow\main_subclass.py

```
from number import Number    # создание подкласса класса C++ в Python (теневой класс)
class MyNumber(Number):
    def add(self, other):
        print 'in Python add...'
        Number.add(self, other)
    def mul(self, other):
        print 'in Python mul...'
        self.data = self.data * other

num = MyNumber(1)            # тот же тест, что в main.cxx
num.add(4)                   # использование подкласса Python теневого класса
num.display()                # add() уточняется в Python
num.sub(2)
num.display()
```

```
num.data = 99
print num.data
num.display()

num.mul(2)                    # mul() реализован в Python
num.display()
del num
```

Теперь вызовы `add` выводят дополнительные сообщения, а `mul` автоматически изменяет данные класса C++ при присваивании `self.data`:

```
[mark@toy ~/...../Integrate/Extend/Swig/Shadow]$ python main_subclass.py
Number: 1
in Python add...
add 4
Number = 5
sub 2
Number = 3
99
Number = 99
in Python mul...
Number = 198
~Number: 198
```

Иными словами, SWIG упрощает использование библиотек классов C++ в качестве базовых классов в сценариях Python. Как обычно, можно импортировать класс C++ интерактивно и немного поэкспериментировать с ним:

```
[mark@toy ~/...../Integrate/Extend/Swig/Shadow]$ python
>>> import numberc
>>> numberc.__file__      # класс C++ плюс сгенерированный связующий модуль
'./numberc.so'
>>> import number         # сгенерированный модуль теневого класса Python
>>> number.__file__
'number.pyc'

>>> x = number.Number(2)  # создать экземпляр класса C++ в Python
Number: 2
>>> y = number.Number(4)  # создать еще один объект C++
Number: 4
>>> x, y
(<C Number instance>, <C Number instance>)

>>> x.display()           # вызов метода C++ (как x->display() в C++)
Number = 2
>>> x.add(y.data)         # получить данные класса C++, вызвать метод C++
add 4
>>> x.display()
Number = 6

>>> y.data = x.data + y.data + 32  # присвоить значение данным класса C++
>>> y.display()                    # y хранит указатель C++ this
Number = 42
```

Так чего же мы добились? В действительности ничего особенного, но если серьезно начать пользоваться SWIG, то самым большим препятствием будет отсутствие на сегодняшний день в SWIG полной поддержки C++. Если в классах используются такие сложные средства C++, как перегрузка операторов и шаблоны, то может потребоваться создать для SWIG упрощенные описания типов классов, а не просто запускать SWIG с исходными заголовочными файлами классов.

Кроме того, используемое в настоящее время в SWIG строковое представление указателей уходит от проблем преобразования и безопасности типов, что в большинстве случаев действует хорошо, но по некоторым сообщениям вызывает осложнения с производительностью и интерфейсами при создании оболочек имеющихся библиотек. Разработка SWIG продолжается, поэтому следует обратиться к руководствам по SWIG и его сайту за деталями, относящимися к этой и другим темам.

Однако в обмен на такие компромиссы SWIG может полностью исключить необходимость в написании связующих слоев для доступа к библиотекам C и C++ из сценариев Python. Если вам доводилось когда-либо писать такой код вручную, вам должно быть ясно, что это *очень* большой выигрыш.

Если же вы пойдете путем работы вручную, обратитесь к стандартным руководствам Python по расширению за дополнительными сведениями по вызовам API, используемым в этой и следующей главах, а также дополнительным инструментам расширений, о которых мы не имеем возможности рассказывать в этой книге. Расширения на C могут лежать в широком диапазоне от коротких входных файлов SWIG до кода, крепко связанного с внутренним устройством интерпретатора Python; практика показывает, что первые из них переносят разрушительное действие времени значительно лучше, чем вторые.

Соединение Python и C++

Стандартная реализация Python в данное время написана на C, поэтому к интерпретатору Python относятся все обычные правила соединения программ C с программами C++. На самом деле в этом контексте Python ничем особым не выделяется, но сделаем несколько замечаний.

При встраивании Python в программу C++ не нужно придерживаться каких-то особых правил. Просто скомпонуйте библиотеку Python и вызывайте ее функции из C++. Заголовочные файлы Python автоматически заключаются в объявления `extern "C" {...}`, чтобы подавить корректировку имен C++. Поэтому библиотека Python выглядит для C++ как любой другой компонент C; нет необходимости перекомпиляции самого Python компилятором C++.

При расширении Python с помощью компонентов C++ заголовочные файлы Python остаются дружественными к C++, поэтому вызовы Python API в расширениях на C++ действуют, как любые другие вызовы из C++ в C. Но необходимо следить, чтобы в коде расширения, видимом Python, использовались объявления `extern "C"`, благодаря чему их сможет вызывать код C, на котором написан Python. Например, для создания оболочки класса C++ SWIG генерирует модуль расширения C++, в котором функция инициализации объявлена таким способом, хотя остальная часть модуля представляет собой чистый C++.

Единственное другое возможное осложнение связано со статическими или глобальными конструкторами объектов C++ при расширении. Если Python (программа C) находится на верхнем уровне системы, такие конструкторы C++ могут не быть выполнены при запуске системы. Такое поведение может зависеть от компилятора, но если ваши объекты C++ не инициализируются при запуске, проверьте, чтобы ваша главная программа компоновалась компилятором C++, а не C.

Если вас интересует интеграция Python/C++ в целом, посмотрите на страницах специальной группы (SIG) C++ на *http://www.python.org* информацию о работе в этой области. Например, система CXX облегчает расширение Python с помощью C++.

20

Встраиваем Python

«Добавить Python. Хорошо перемешать. Повторить»

В предыдущей главе мы изучили одну половину интеграции Python с C – вызов сервисов C из Python. Этот способ позволяет программистам ускорять выполнение программ, переводя их на C, и использовать внешние библиотеки, создавая для них оболочки в виде модулей и типов расширения на C. Но столь же полезным может оказаться и обратное – вызов Python из C. Путем передачи отдельных компонентов приложения встроенному коду Python мы даем возможность изменять их на месте, не поставляя заказчику весь код системы.

В данной главе рассказывается об этой второй стороне интеграции Python с C. Здесь говорится об интерфейсах C к Python, позволяющих программам, написанным на C-совместимых языках, выполнять код программ Python. В этом режиме Python выступает в качестве встроенного управляющего языка (или, как иногда говорят, «макроязыка»). Хотя встраивание представляется здесь, по большей части, изолированно, нужно помнить, что лучше всего рассматривать поддержку интеграции в Python как целое. Структура системы обычно определяет соответствующий подход к интеграции: расширения C, встроенные вызовы кода или то и другое вместе. Завершается глава обсуждением нескольких крупных платформ интеграции, таких как COM и JPython, которые предоставляют более широкие возможности интеграции компонентов.

Обзор API встраивания в C

Первое, что следует отметить в API встроенных вызовов Python, это меньшую его структурированность, чем у интерфейсов расширения. Для встраивания Python в C может потребоваться более творческий подход, чем при расширении; программист должен выбирать средства для реализации интеграции с Python из общей совокупности вызовов, а не писать код согласно стереотипной структуре. Положительной стороной такой свободной структуры является то, что можно объединять в программах встроенные вызовы и стратегии, создавая произвольные архитектуры интеграции.

Отсутствие строгой модели встраивания в значительной мере является результатом менее четко обозначенных целей. При *расширении* Python есть четкое разделение ответственности между Python и C и ясная структура интеграции. Модули и типы C должны соответствовать модели модулей/типов Python путем соблюдения стандартных структур расширений. В результате интеграция оказывается незаметной для клиентов Python: расширения C выглядят, как объекты Python, и выполняют большую часть работы.

Но при *встраивании* Python структура не так очевидна; так как внешним уровнем является C, не совсем ясно, какой модели должен придерживаться встроенный код Ру-

thon. В C может потребоваться выполнять загружаемые из модулей объекты, загружаемые из файлов или выделенные в документах строки и т. д. Вместо того чтобы решать, чего можно и чего нельзя делать в C, Python предоставляет набор общих средств интерфейса встраивания, применяемых и организуемых согласно целям встраивания.

Большинство этих средств соответствует тем средствам, которые доступны программам Python. В табл. 20.1 перечислены некоторые наиболее часто встречающиеся вызовы API, используемые для встраивания, и их эквиваленты в Python. В целом, если можно установить, как решить задачи встраивания с помощью чистого кода Python, то, вероятно, найдутся средства C API, позволяющие достичь таких же результатов.

Таблица 20.1. Часто используемые функции API

Вызов C API	Эквивалент Python
PyImport_ImportModule	import module, __import__
PyImport_ReloadModule	reload(module)
PyImport_GetModuleDict	sys.modules
PyModule_GetDict	module.__dict__
PyDict_GetItemString	dict[key]
PyDict_SetItemString	dict[key]=val
PyDict_New	dict = {}
PyObject_GetAttrString	getattr(obj, attr)
PyObject_SetAttrString	setattr(obj, attr, val)
PyEval_CallObject	apply(funcobj, argstuple)
PyRun_String	eval(exprstr), exec stmtstr
PyRun_File	execfile(filename)

Так как встраивание основывается на выборе вызова API, знакомство с Python C API необходимо для решения задачи встраивания. В этой главе представлен ряд характерных примеров встраивания и обсуждаются стандартные вызовы API, но нет полного списка всех имеющихся в нем инструментов. Разобравшись с приведенными примерами, возможно, потребуется обратиться к руководствам Python по интеграции за дополнительными сведениями о том, какие вызовы есть в этой области. В последней версии Python есть два стандартных руководства для программистов, занимающихся интеграцией с C/C++: *Extending and Embedding*, учебник по интеграции, и *Python/C API*, справочник по библиотеке окружения исполнения Python.

Эти руководства можно найти на прилагаемом к книге CD или загрузить их последние версии с *http://www.python.org*. Помимо данной главы лучшим источником свежей и полной информации по средствам Python API послужат эти два руководства.

Что представляет собой встроенный код?

Прежде чем переходить к деталям, разберемся с базовыми понятиями встраивания. Когда в этой книге говорится о «встроенном» коде Python, имеется в виду любая программная структура Python, которая может быть выполнена из C. Вообще говоря, встроенный код Python может быть нескольких видов:

Строки кода

Программы на C могут представлять программы Python в виде символьных строк и выполнять их как выражения или операторы (подобно `eval` и `exec`).

Вызываемые объекты

Программы на C могут загружать или обращаться к вызываемым объектам Python, таким как функции, методы и классы, и вызывать их со списками аргументов (как `apply`).

Файлы с кодом

Программы на C могут выполнять целые программные файлы Python, импортируя модули и выполняя файлы сценариев через API или общие системные вызовы (например, `popen`).

Физически в программу C обычно встраивается двоичная библиотека Python; фактически выполняемый из C код Python может поступать из разнообразных источников:

- *Строки кода* могут загружаться из файлов, быть получены из баз данных и полок, выделены из файлов HTML или XML, читаться через сокеты, строиться или быть прошитыми в программах C, передаваться функциям расширения C из кода регистрации Python и т. д.
- *Вызываемые объекты* могут загружаться из модулей Python, возвращаться другими вызовами Python API, передаваться функциям расширения C из кода регистрации Python и т. д.
- *Файлы кода* просто существуют в виде файлов, модулей и выполняемых сценариев.

Регистрация является приемом, часто используемым при организации обратных вызовов, который более подробно будет изучен далее в этой главе. Но что касается строк кода, возможных источников существует столько, сколько их есть для строк символов C. Например, программы на C могут динамически строить произвольный код Python, создавая и выполняя строки.

Наконец, когда получен некоторый код Python, который должен быть выполнен, необходим какой-то способ связи с ним: код Python может потребовать входные данные, передаваемые из слоя C, и может создать вывод для передачи результатов обратно в C. На самом деле встраивание в целом представляет интерес, когда у встроенного кода есть доступ к содержащему его слою C. Обычно средство связи предполагается видом встроенного кода:

- *Строки кода*, являющиеся выражениями Python, возвращают *значение выражения* в качестве выходных данных. Как входные, так и выходные данные могут иметь вид *глобальных переменных* в том пространстве имен, в котором выполняется строка кода – C может устанавливать значения переменных, служащих входными данными, выполнять код Python и получать переменные с результатом выполнения кода. Входные и выходные данные можно также передавать с помощью экспортируемых *вызовов расширений* на C – код Python может с помощью модулей и типов C получать или устанавливать переменные в охватывающем слое C. Схемы связи часто являются комбинированными; например, C может заранее назначать глобальные имена объектам, экспортирующим вызовы состояния и интерфейса во встроенный код Python.[1]

[1] Если нужен пример, вернитесь к обсуждению Active Scripting в главе 15 «Более сложные темы Интернета». В этой системе загружается код Python, встроенный в файл HTML веб-страницы, в пространстве имен глобальным переменным присваиваются объекты, предоставляющие доступ к окружению веб-броузера, и выполняется код Python в пространстве имен, где осуществлено назначение объектам. Недавно я работал над проектом, в котором делалось нечто схожее, но код Python был встроен в документы XML, а объекты, присваиваемые глобальным переменным в пространстве имен кода, представляли собой графические элементы в GUI.

- *Вызываемые объекты* могут получать входные данные в виде *аргументов* функции и создавать результаты в виде *возвращаемых значений* функции. Переданные изменяемые аргументы (например, списки, словари, экземпляры классов) можно использовать во встроенном коде одновременно для ввода и вывода – сделанные в Python изменения сохраняются в объектах, которыми владеет C. Для связи с C объекты могут также использовать технику глобальных переменных и интерфейса расширений C, описанную для строк.
- *Файлы кода* по большей части могут использовать для связи такую же технику, как строки кода; при выполнении в качестве отдельных программ файлы могут также применять технику IPC.

Естественно, все виды встроенного кода могут обмениваться данными с C, используя общие средства системного уровня: файлы, сокеты, каналы и т. д. Однако обычно эти способы действуют медленнее и менее непосредственно.

Основные приемы встраивания кода

Как можно заключить из предшествующего обзора, встраивание предоставляет большую гибкость. Чтобы проиллюстрировать стандартные приемы встраивания в действии, в этом разделе представлен ряд коротких программ на C, которые в том или ином виде выполняют код Python. Большинство этих примеров используют простой файл модуля Python, показанный в примере 20.1.

Пример 20.1. PP2E\Integrate\Embed\Basics\usermod.py

```
##########################################################################################
# C выполняет код Python этого модуля в режиме встраивания. Такой файл можно редактировать,
# не трогая слоя C. Это стандартный код Python (преобразования делает C).
# Можно выполнять код стандартных модулей, например string.
##########################################################################################

import string

message = 'The meaning of life...'

def transform(input):
    input = string.replace(input, 'life', 'Python')
    return string.upper(input)
```

Если вы хоть сколько-нибудь знакомы с Python, то определите, что этот файл определяет строку и функцию; функция возвращает переданную ей строку после выполнения замены в строке и перевода ее в верхний регистр. Из Python пользоваться модулем просто:

```
[mark@toy ~/.../PP2E/Integrate/Embed/Basics]$ python
>>> import usermod                                  # импорт модуля
>>> usermod.message                                 # получить строку
'The meaning of life...'
>>> usermod.transform(usermod.message)              # вызвать функцию
'THE MEANING OF PYTHON...'
```

При надлежащем использовании API ненамного сложнее точно так же использовать этот модуль в C.

Выполнение простых строк кода

По-видимому, проще всего выполнить код Python из C, вызвав функцию API `PyRun_SimpleString`. С ее помощью программы C могут выполнять программы Python, представленные в виде массивов символьных строк C. Этот вызов весьма ограничен: весь код выполняется в одном и том же пространстве имен (модуль `__main__`), строки кода должны быть операторами Python (не выражениями), отсутствует простой способ обмена входными и выходными данными с выполняемым кодом Python. Тем не менее это простой способ для начала; программа C примера 20.2 выполняет код Python и получает те же результаты, что и в интерактивном сеансе, приведенном в предыдущем разделе.

Пример 20.2. PP2E\Integrate\Embed\Basics\embed-simple.c

```
/*****************************************************
 * простые строки кода: C действует как интерактивное
 * приглашение, код выполняется в __main__, вывод не посылается в C;
 *****************************************************/

#include <Python.h>    /* standard API def */

main() {
    printf("embed-simple\n");
    Py_Initialize();
    PyRun_SimpleString("import usermod");                /* загрузка файла .py */
    PyRun_SimpleString("print usermod.message");         /* в пути поиска python */
    PyRun_SimpleString("x = usermod.message");           /* компилировать и выполнить */
    PyRun_SimpleString("print usermod.transform(x)");
}
```

Во-первых, нужно обратить внимание на то, что при встраивании Python программы C всегда вызывают `Py_Initialize`, чтобы инициализировать подключаемые библиотеки Python, прежде чем использовать какие-либо другие функции API. Остальная часть кода проста – C передает Python готовые строки, примерно совпадающие с тем, что вводилось интерактивно. Внутренне `PyRun_SimpleString` вызывает для выполнения переданных из C строк компилятор и интерпретатор Python; компилятор Python, как обычно, всегда есть в системах, где установлен Python.

Компиляция и выполнение

Чтобы создать самостоятельно выполняемую программу из этого файла исходного кода C, необходимо скомпоновать результат его компиляции с файлом библиотеки Python. В данной главе «библиотека» обычно означает двоичный библиотечный файл (например, файл *.a* в Unix), создаваемый при компиляции Python, а не библиотеку исходного кода Python.

На сегодняшний день все, что касается Python и может потребоваться в C, компилируется в один библиотечный файл *.a* при сборке интерпретатора. Функция программы `main` поступает из вашего кода C, а в зависимости от расширений, установленных для Python, может потребоваться компоновка внешних библиотек, к которым обращается библиотека Python.

В предположении, что никаких дополнительных библиотек расширения не требуется, пример 20.3 представляет минимальный make-файл для Linux, с помощью которого собирается программа на C из примера 20.2. Как всегда, детали make-файла зависят от платформы, поэтому ищите указаний в руководствах по Python. Данный make-файл использует путь включаемых файлов Python при поиске *Python.h* на этапе компиляции и добавляет файл библиотеки Python на конечном этапе компоновки, чтобы сделать возможными вызовы API в программе на C.

Пример 20.3. PP2E\Integrate\Embed\Basics\makefile.

```
# make-файл linux, строящий выполняемую программу на C со встроенным Python,
# в предположении, что не требуется компоновка с библиотеками внешних модулей;
# использует заголовочные файлы Python, подключает файл библиотеки Python;
# те и другие могут быть на вашей машине в других каталогах (например, /usr);
# установите MYPY согласно вашему дереву установки Python, измените версию lib;

PY    = $(MYPY)
PYLIB = $(PY)/libpython1.5.a
PYINC = -I$(PY)/Include -I$(PY)

embed-simple: embed-simple.o
    gcc embed-simple.o $(PYLIB) -g -export-dynamic -lm -ldl -o embed-simple

embed-simple.o: embed-simple.c
    gcc embed-simple.c -c -g $(PYINC)
```

На практике все может оказаться не так просто, и понадобится терпение, чтобы добиться результата. В действительности я использовал make-файл примера 20.4 для сборки всех программ C этого раздела под Linux.

Пример 20.4. PP2E\Integrate\Embed\Basics\makefile.basics

```
# собрать все 5 примеров основ встраивания с подключением библиотек внешних модулей;
# при необходимости source setup-pp-embed.csh

PY    = $(MYPY)
PYLIB = $(PY)/libpython1.5.a
PYINC = -I$(PY)/Include -I$(PY)

LIBS = -L/usr/lib \
       -L/usr/X11R6/lib \
       -lgdbm -ltk8.0 -ltcl8.0 -lX11 -lm -ldl

BASICS = embed-simple embed-string embed-object embed-dict embed-bytecode

all:$(BASICS)

embed%: embed%.o
    gcc embed$*.o $(PYLIB) $(LIBS) -g -export-dynamic -o embed$*

embed%.o: embed%.c
    gcc embed$*.c -c -g $(PYINC)

clean:
    rm -f *.o *.pyc $(BASICS) core
```

В этой версии подключаются библиотеки Tkinter, потому что используемая библиотека Python была собрана с включением Tkinter. Ваша библиотека Python может потребовать подключения значительно большего числа внешних библиотек, и, честно говоря, проследить все зависимости компоновщика может оказаться утомительным. Необходимые библиотеки могут зависеть от платформы и установки Python, поэтому я мало чем могу помочь для облегчения этого процесса (в конце концов, это C).

Но если вы будете много заниматься встраиванием, то можно скомпилировать Python на своей машине из исходного кода, отключив все ненужные расширения в файле `Modules/Setup`. В результате будет создана библиотека Python с минимумом внешних зависимостей, компоновать программы с которой значительно легче. Например, если ваш встраиваемый код не будет строить GUI, можно просто удалить Tkinter из библиотеки; смотрите подробности в файле `Setup`. Список внешних библиотек, к которым обращается Python, можно найти в генерируемых make-файлах, находящихся в дере-

ве исходного кода Python. В любом случае решить проблему с зависимостями при компоновке нужно лишь один раз.

Получив рабочий make-файл, запускайте его для сборки программ C с подключенными библиотеками Python. Результирующая программа на C запускается как обычно:[1]

```
[mark@toy ~/.../PP2E/Integrate/Embed/Basics]$ embed-simple
embed-simple
The meaning of life...
THE MEANING OF PYTHON...
```

Этот вывод производят в основном операторы Python `print`, которые C посылает присоединенной библиотеке Python. Это похоже на то, как если бы C стал интерактивным программистом Python.

Однако строки Python, выполняемые C, вероятно, не стоит жестко кодировать в программе C. Их можно загружать из текстового файла, извлекать из файлов HTML или XML, получать из постоянной базы данных или через сокеты и т. п. При использовании таких внешних источников строки кода Python, выполняемые из C, можно произвольно изменять без необходимости перекомпиляции выполняющей их программы C. Их даже могут изменять на месте конечные пользователи системы. Чтобы использовать строки кода с максимальной пользой, мы должны перейти к более гибким инструментам API.

Выполнение строк кода с использованием результатов и пространств имен

В примере 20.5 используются следующие вызовы API для выполнения строк кода, возвращающих значения выражений в C:

- `Py_Initialize` инициализирует, как и раньше, присоединенные библиотеки Python
- `PyImport_ImportModule` импортирует модуль Python, возвращает указатель на него
- `PyModule_GetDict` загружает объект словаря атрибутов модуля
- `PyRun_String` выполняет строку кода в явных пространствах имен
- `PyObject_SetAttrString` выполняет присвоение атрибуту объекта по строке имени
- `PyArg_Parse` преобразует объект возвращаемого значения Python в формат C

С помощью вызовов импорта загружается пространство имен модуля `usermod`, показанного в примере 20.1, чтобы прямо выполнять в нем строки кода (обращаясь к именам, определенным в этом модуле, не квалифицируя их). `Py_Import_ImportModule` имеет сходство с оператором Python `import`, но импортированный объект модуля возвращается в C, а не присваивается переменной Python. Поэтому он скорее похож на встроенную функцию Python `__import__`, использовавшуюся в примере 7.32.

---

[1] Моя среда компиляции несколько индивидуальна (в самом деле, необычна), поэтому сначала я должен выполнить `source $PP2E/Config/setup-pp-embed.csh`, чтобы PYTHONPATH указывала на каталог библиотеки исходного кода особой установки Python на моей машине. По крайней мере в Python 1.5.2. могут возникнуть проблемы с обнаружением каталогов стандартной библиотеки Python при встраивании, особенно если на одной и той же машине есть несколько установок Python (например, могут не совпадать версии интерпретатора и библиотеки). В вашей среде компиляции таких проблем может не возникнуть, но посмотрите содержимое указанного файла, если при попытке выполнения программ на C со встроенным Python возникают ошибки начального запуска. Если вы засунули Python куда-то в темный угол, может потребоваться настроить свои сценарии регистрации или выполнить такой файл настройки, прежде чем запускать примеры со встраиванием.

Фактическим выполнением кода здесь занимается вызов PyRun_String. Он принимает строку кода, флаг режима синтаксического разбора и указатели на объекты словарей, которые будут служить в качестве глобального и локального пространств имен при выполнении строки кода. Флаг режима может иметь значение Py_eval_input для выполнения выражения или Py_file_input для выполнения оператора; при выполнении выражения этот вызов возвращает результат вычисления выражения (в виде указателя на объект PyObject*). Два аргумента с указателями на словари пространств имен позволяют различать глобальную и локальную области видимости, но обычно они задают один и тот же словарь, благодаря чему код выполняется в одном пространстве имен.[1]

Пример 20.5. PP2E\Integrate\Embed\Basics\embed-string.c

```
/* строки кода с результатами и пространствами имен */

#include <Python.h>

main() {
    char *cstr;
    PyObject *pstr, *pmod, *pdict;
    printf("embed-string\n");
    Py_Initialize();

    /* получить usermod.message */
    pmod  = PyImport_ImportModule("usermod");
    pdict = PyModule_GetDict(pmod);
    pstr  = PyRun_String("message", Py_eval_input, pdict, pdict);

    /* преобразовать в C */
    PyArg_Parse(pstr, "s", &cstr);
    printf("%s\n", cstr);

    /* присвоить usermod.X */
    PyObject_SetAttrString(pmod, "X", pstr);

    /* вывести usermod.transform(X) */
    (void) PyRun_String("print transform(X)", Py_file_input, pdict, pdict);
    Py_DECREF(pmod);
    Py_DECREF(pstr);
}
```

После компиляции и выполнения этого файла получается такой же вывод, как у его предшественника:

```
[mark@toy ~/.../PP2E/Integrate/Embed/Basics]$ embed-string
embed-string
The meaning of life...
THE MEANING OF PYTHON...
```

Но здесь происходят совершенно иные действия. На этот раз C загружает, преобразует и выводит значение атрибута message модуля Python путем непосредственного выполнения строкового выражения и присваивает значение глобальной переменной (X) в пространстве имен модуля, служащей в качестве входных данных строки оператора Python print.

---

[1] Родственная функция позволяет выполнять *файлы* с кодом, но в этой главе она не показана: PyObject* PyRunFile(FILE *fp, char *filename, mode, globals, locals). Поскольку всегда можно загрузить файл текста и выполнить его как одну строку с помощью PyRun_String, в вызове PyRun_File не всегда есть необходимость. В таких многострочных массивах символов строчки завершаются \n, а отступы создают блоки как обычно.

Благодаря тому, что вызов выполнения строки в этой версии позволяет задавать пространства имен, можно лучше расчленить встроенный код, выполняемый системой – можно иметь раздельные пространства имен для разных групп, чтобы избежать изменения переменных в других группах. А так как этот вызов возвращает результат, облегчается связь со встроенным кодом – результаты выражений служат выходными данными, а присваивания глобальным переменным в пространстве имен, в котором выполняется код, служат в качестве входных данных.

Прежде чем двинуться дальше, я должен остановиться на двух проблемах, возникающих здесь при написании кода. Во-первых, эта программа также декрементирует *счетчик ссылок* для объектов, передаваемых ей из Python, с помощью вызова `PyDECREF`, представленного в главе 19 «Расширение Python». Эти вызовы здесь не особенно требуются (память объектов все равно освобождается при выходе из программ), но они демонстрируют, как интерфейсы встраивания должны управлять счетчиками ссылок, когда Python передает владение ими в C. Если бы, скажем, это была функция, вызываемая из более крупной системы, обычно потребовалось бы уменьшить счетчик, чтобы позволить Python освободить память объектов.

Во-вторых, в настоящей программе следует сразу проверять значения, возвращаемые *всеми* вызовами API, чтобы обнаруживать ошибки (например, неудачу при импорте). Для упрощения кода проверки ошибок опущены в примере этого раздела, но в последующих листингах кода они будут, и необходимо включать их в код создаваемых программ для повышения устойчивости.

Вызов объектов Python

В двух последних разделах мы занимались выполнением строк кода, но программы C могут также легко работать с объектами Python. Пример 20.6 выполняет ту же задачу, что и примеры 20.2 и 20.5, но использует другие средства API для прямого взаимодействия с объектами модуля Python:

- `PyImport_ImportModule` импортирует модуль в C, как и прежде
- `PyObject_GetAttrString` загружает значение атрибута объекта по имени
- `PyEval_CallObject` вызывает функцию Python (или класс, или метод)
- `PyArg_Parse` преобразует объекты Python в значения C
- `Py_BuildValue` преобразует значения C в объекты Python

С обеими функциями преобразования мы познакомились в предыдущей главе. Вызов `PyEvalCallObject` в этой версии является ключевым: он выполняет импортированную функцию, передавая ей кортеж аргументов подобно встроенной функции Python `apply`. Значение, возвращаемое функцией Python, поступает в C в виде `PyObject*`, указателя на общий объект Python.

Пример 20.6. PP2E\Integrate\Embed\Basics\embed-object.c

```
/* загрузка и вызов объектов из модулей */

#include <Python.h>

main() {
    char *cstr;
    PyObject *pstr, *pmod, *pfunc, *pargs;
    printf("embed-object\n");
    Py_Initialize();

    /* получить usermod.message */
    pmod = PyImport_ImportModule("usermod");
```

```
    pstr = PyObject_GetAttrString(pmod, "message");

    /* преобразовать строку в C */
    PyArg_Parse(pstr, "s", &cstr);
    printf("%s\n", cstr);
    Py_DECREF(pstr);

    /* вызвать usermod.transform(usermod.message) */
    pfunc = PyObject_GetAttrString(pmod, "transform");
    pargs = Py_BuildValue("(s)", cstr);
    pstr  = PyEval_CallObject(pfunc, pargs);
    PyArg_Parse(pstr, "s", &cstr);
    printf("%s\n", cstr);

    /* освободить объекты */
    Py_DECREF(pmod);
    Py_DECREF(pstr);
    Py_DECREF(pfunc);        /* действительной необходимости в main() нет,*/
    Py_DECREF(pargs);        /* так как освобождается вся память  */
}
```

После компиляции и выполнения получается тот же результат:

```
[mark@toy ~/.../PP2E/Integrate/Embed/Basics]$ embed-object
embed-object
The meaning of life...
THE MEANING OF PYTHON...
```

Но на этот раз весь вывод создается средствами C – сначала путем загрузки значения атрибута `message` модуля Python, а затем путем прямой загрузки и вызова объекта функции модуля `transform` и вывода возвращаемого им значения, пересылаемого в C. Входными данными функции `transform` является аргумент функции, а не глобальная переменная, которой предварительно присвоено значение. Обратите внимание, что на этот раз `message` загружается как атрибут модуля, а не в результате выполнения ее имени как строки кода; часто есть несколько способов получить один и тот же результат, используя разные вызовы API.

Показанное здесь выполнение функций в модулях дает простой способ организации встраивания; код в файле модуля можно произвольно менять, не перекомпилируя выполняющую его программу на C. Обеспечивается также модель прямой связи: входные и выходные данные кода Python могут иметь вид аргументов функции и возвращаемых ей значений.

Выполнение строк в словарях

Выше, при запуске с помощью `PyRun_String` выражений с результатами, выполнение кода происходило в пространстве имен существующего модуля Python. Однако иногда для выполнения строк кода удобнее создавать совершенно новое пространство имен, независимое от существующих файлов модулей. Файл C в примере 20.7 показывает, как это делается; пространство имен создается как новый объект словаря Python, при этом в процессе участвует ряд новых вызовов API:

- `PyDict_New` создает новый пустой объект словаря
- `PyDict_SetItemString` выполняет присваивание по ключу словаря
- `PyDict_GetItemString` загружает значение из словаря по ключу
- `PyRun_String` выполняет строку кода в пространствах имен, как и ранее
- `PyEval_GetBuiltins` получает модуль с областью видимости встроенных элементов

Главная хитрость здесь заключается в новом словаре. Входные и выходные данные для строк встроенного кода отображаются в этот словарь при его передаче в качестве словарей пространств имен кода в вызове PyRun_String. В итоге программа на C из примера 20.7 работает в точности, как следующий код на Python:

```
>>> d = {}
>>> d['Y'] = 2
>>> exec 'X = 99' in d, d
>>> exec 'X = X + Y' in d, d
>>> print d['X']
101
```

Но здесь каждая операция Python заменяется вызовом C API.

Пример 20.7. PP2E\Integrate\Embed\Basics\embed-dict.c

```
/***************************************************
 * создание нового словаря для пространства имен строки кода;
 ***************************************************/

#include <Python.h>

main() {
    int cval;
    PyObject *pdict, *pval;
    printf("embed-dict\n");
    Py_Initialize();

    /* make a new namespace */
    pdict = PyDict_New();
    PyDict_SetItemString(pdict, "__builtins__", PyEval_GetBuiltins());

    PyDict_SetItemString(pdict, "Y", PyInt_FromLong(2));    /* dict['Y'] = 2   */
    PyRun_String("X = 99",  Py_file_input, pdict, pdict);  /* выполнить операторы */
    PyRun_String("X = X+Y", Py_file_input, pdict, pdict);  /* те же X и Y     */
    pval = PyDict_GetItemString(pdict, "X");               /* получить dict['X'] */

    PyArg_Parse(pval, "i", &cval);                         /* преобразовать в C */
    printf("%d\n", cval);                                  /* result=101 */
    Py_DECREF(pdict);
}
```

После компиляции и выполнения эта программа C выводит следующее:

```
[mark@toy ~/.../PP2E/Integrate/Embed/Basics]$ embed-dict
embed-dict
101
```

На этот раз вывод иной: он отражает значение переменной Python X, присвоенное встроенными строками кода Python и полученное в C. В целом C может получать атрибуты модуля, либо вызывая с модулем PyObject_GetAttrString, либо обращаясь к словарю атрибутов модуля с помощью PyDict_GetItemString (строки выражений тоже работают, но не так прямо). В данном случае модуля нет вообще, поэтому для обращения к пространству имен кода в C используется доступ к словарю по индексу.

Помимо предоставления возможности расчленения пространств имен кодовых строк, независимого от файлов модулей Python базовой системы, эта схема предоставляет естественный механизм связи. Значения, сохраняемые в новом словаре перед выполнением кода, служат входными данными, а имена, которым производится присвоение встроенным кодом, впоследствии могут извлекаться из словаря в качестве выходных данных кода. Например, переменная Y во второй строке обращается к имени,

которому в C присваивается значение 2; значение X присваивается кодом Python и получается позднее в C как выведенный результат.

Здесь есть один прием, требующий пояснения. В каждом пространстве имен модуля Python есть ссылка на пространство имен области видимости встроенных объектов, где находятся такие имена, как open или len. На самом деле это та ссылка, по которой следует Python на последнем этапе своей процедуры поиска имен в локальной/глобальной/встроенной области видимости.[1] На сегодняшний день встроенный код отвечает за установку ссылки на область видимости _builtins_ в словарях, служащих пространствами имен. Python автоматически устанавливает эту ссылку во всех других пространствах имен, в которых производится выполнение кода, и это требование к встраиванию может быть снято в будущем (оно выглядит излишне магическим, чтобы долгое время быть необходимым). А пока просто делайте то же, что в этом примере, для инициализации ссылки на встроенные элементы в словарях, создаваемых для выполнения кода в C.

Предварительная компиляция строк в байт-код

При вызове объектов функций Python из C вы фактически выполняете откомпилированный байт-код, связанный с объектом (например, телом функции). При выполнении строк Python должен скомпилировать строку перед ее выполнением. Так как компиляция является медленным процессом, это может стать существенным лишним расходом при многократном выполнении строки. Чтобы избежать его, откомпилируйте строку в объект байт-кода, который будет выполняться позднее, с помощью вызовов API, показанных в примере 20.8:[2]

- Py_CompileString компилирует строку кода, возвращает объект байт-кода
- PyEval_EvalCode выполняет объект, скомпилированный в байт-код

Первый из этих вызовов принимает флаг режима, обычно передаваемый в PyRun_String, и второй строковый аргумент, используемый только в сообщениях об ошибках. Второй вызов принимает два словаря пространств имен. Эти два вызова API использованы в примере 20.8 для компиляции и исполнения трех строк кода Python.

Пример 20.8. PP2E\Integrate\Embed\Basics\embed-bytecode.c

```
/* предварительная компиляция строк кода в объекты байт-кода */

#include <Python.h>
#include <compile.h>
#include <eval.h>

main() {
    int i;
    char *cval;
    PyObject *pcode1, *pcode2, *pcode3, *presult, *pdict;
    char *codestr1, *codestr2, *codestr3;
```

[1] Эта ссылка участвует также в режиме ограниченного выполнения Python, описанном в главе 15. Переключая ссылку с области видимости встроенных переменных на модуль с ограниченными наборами атрибутов и специальными версиями встроенных вызовов типа open, модуль rexec может управлять доступом к машине из кода, выполняемого через его интерфейс.

[2] На случай, если вы уже заглянули вперед: в текущей стандартной реализации Python *байт-код* является промежуточным представлением откомпилированного кода программы. Это низкоуровневый двоичный формат, который может быстро интерпретироваться системой выполнения Python. Байт-код обычно автоматически генерируется при импорте модуля, но при выполнении исходных строк из C может и речи не идти об импорте.

```
    printf("embed-bytecode\n");

    Py_Initialize();
    codestr1 = "import usermod\nprint usermod.message";     /* statements */
    codestr2 = "usermod.transform(usermod.message)";        /* expression */
    codestr3 = "print '%d:%d' % (X, X ** 2),";              /* use input X */

    /* создать новый словарь пространства имен */
    pdict = PyDict_New();
    if (pdict == NULL) return -1;
    PyDict_SetItemString(pdict, "__builtins__", PyEval_GetBuiltins());

    /* precompile strings of code to bytecode objects */
    pcode1 = Py_CompileString(codestr1, "<embed>", Py_file_input);
    pcode2 = Py_CompileString(codestr2, "<embed>", Py_eval_input);
    pcode3 = Py_CompileString(codestr3, "<embed>", Py_file_input);

    /* выполнить откомпилированный байт-код в пространстве имен словаря */
    if (pcode1 && pcode2 && pcode3) {
        (void)    PyEval_EvalCode((PyCodeObject *)pcode1, pdict, pdict);
        presult = PyEval_EvalCode((PyCodeObject *)pcode2, pdict, pdict);
        PyArg_Parse(presult, "s", &cval);
        printf("%s\n", cval);
        Py_DECREF(presult);

        /* повторное выполнение объекта кода */
        for (i = 0; i <= 10; i++) {
            PyDict_SetItemString(pdict, "X", PyInt_FromLong(i));
            (void) PyEval_EvalCode((PyCodeObject *)pcode3, pdict, pdict);
        }
        printf("\n");
    }

    /* освободить память объектов */
    Py_XDECREF(pdict);
    Py_XDECREF(pcode1);
    Py_XDECREF(pcode2);
    Py_XDECREF(pcode3);
}
```

В этой программе объединен ряд приемов, с которыми мы уже встречались. Например, пространством имен, в котором выполняются скомпилированные строки кода, является вновь создаваемый словарь (а не существующий объект модуля), а входные данные для строк кода передаются как предустановленные переменные в пространстве имен. После компиляции и выполнения первая часть вывода аналогична предыдущим примерам этого раздела, но последняя строчка представляет 11-кратное выполнение той же, предварительно откомпилированной строки кода:

```
[mark@toy ~/.../PP2E/Integrate/Embed/Basics]$ embed-bytecode
embed-bytecode
The meaning of life...
THE MEANING OF PYTHON...
0:0 1:1 2:4 3:9 4:16 5:25 6:36 7:49 8:64 9:81 10:100
```

Если ваша система выполняет строки несколько раз, такое предварительное компилирование в байт-код влечет значительное ускорение.

Регистрация объектов для обработки обратных вызовов

В примерах, приведенных до сего момента, C выполнялся и вызывал код Python из обычной последовательности команд основной программы. Однако программы не всегда работают таким образом; в некоторых случаях программы создаются на основе архитектуры *управления событиями*, в которой код выполняется только в ответ на те или иные события. Событием может являться щелчок конечным пользователем по кнопке в GUI, отправка сигнала операционной системой или просто выполнение программой действий, ассоциируемых с вводом данных в таблицу.

Так или иначе, программный код в такой архитектуре обычно организуется в виде обработчиков обратных вызовов (*callback handlers*) – фрагментов кода, выполнение которых организовано логикой, обрабатывающей события. Для реализации обработчиков обратного вызова в такой системе легко использовать встроенный код Python; в действительности для запуска обработчиков Python слой обработки событий может просто использовать средства API вызова встроенного кода, с которыми мы познакомились в этой главе.

В этой модели единственный новый прием состоит в том, чтобы сообщить слою C о том, какой код должен выполняться для каждого события. Обработчики каким-то образом должны быть зарегистрированы в C для связи с возможными событиями. В целом существует ряд способов добиться такой связи между кодом и событиями; например, программы C могут:

- Загружать и вызывать *функции* по имени события из одного или нескольких файлов *модулей*
- Загружать и выполнять *строки* кода, ассоциированные с именами событий, из *базы данных*
- Загружать и выполнять код, ассоциируемый с *тегами* событий в HTML или XML[1]
- Выполнять код Python, который обращается к C с запросом о том, что должно быть выполнено

И так далее. В действительности механизмом регистрации обратного вызова может стать все, что каким-то образом ассоциирует объекты или строки с идентификаторами. У некоторых из таких приемов есть свои преимущества. Например, получение кода обратного вызова из файлов модулей поддерживает динамическую перегрузку (как было показано в главе 9 «Примеры больших GUI», `reload` действует в отношении модулей и не обновляет объекты, хранящиеся непосредственно). При этом ни в одной из трех первых схем не требуется писать специальные программы Python, занимающиеся только регистрацией обработчиков для последующего выполнения.

Однако чаще, по-видимому, для регистрации обработчиков обратного вызова используется последний подход: регистрация обработчиков в C кодом Python при обратном вызове C через интерфейс расширений. Хотя в этой схеме есть свои минусы, она предоставляет естественную и прямую модель в ситуациях, где обратные вызовы ассоциируются с большим числом объектов.

Рассмотрим, например, GUI, построенный в виде дерева объектов графических элементов в сценарии Python. Если с каждым объектом дерева может быть связан обработ-

[1] А если компилятор C решит, то может даже выполнить встроенный код Python, использующий стандартные средства Python обработки HTML и XML для анализа встроенного кода, ассоциированного с тегом события. Смотрите подробнее об этих парсерах в руководстве по библиотеке Python.

чик события, проще будет регистрировать обработчики при вызове методов графических элементов, образующих дерево. При связывании обработчиков с объектами графических элементов в отдельной структуре, такой как файл модуля или файл HTML, потребуется дополнительная работа с перекрестными ссылками, чтобы поддерживать соответствие обработчиков дереву.[1]

В следующих файлах C и Python демонстрируются основы техники программирования, используемой при реализации явно регистрируемых обработчиков обратных вызовов. Файл C примера 20.9 реализует интерфейсы регистрации обработчиков Python, а также код для запуска этих обработчиков в ответ на события:

Маршрутизатор событий

Функция `Route_Event` реагирует на событие, вызывая объект функции Python, ранее переданный из Python в C.

Регистрация обратного вызова

Функция `Register_Handler` сохраняет переданный ей указатель на объект функции Python в глобальной переменной C. Python вызывает `Register_Handler` через простой модуль расширения C `cregister`, создаваемый этим файлом.

Генерация событий

Для моделирования реальных событий можно вызвать через созданный модуль C функцию Python `Trigger_Event`, которая сгенерирует событие.

Иными словами, в этом примере используются оба знакомых нам интерфейса – расширения и встраивания – для вызова кода Python обработчика события.

Пример 20.9. PP2E\Integrate\Mixed\Regist\cregister.c

```
#include <Python.h>
#include <stdlib.h>

/*********************************************/
/* 1) код маршрутизации событий объекту Python */
/* можно было бы выполнять здесь строки        */
/*********************************************/

static PyObject *Handler = NULL;                /* хранить объект Python в C */

void Route_Event(char *label, int count)
{
    char *cres;
    PyObject *args, *pres;

    /* call Python handler */
    args = Py_BuildValue("(si)", label, count);  /* создать список аргументов */
    pres = PyEval_CallObject(Handler, args);     /* применить: выполнить вызов */
    Py_DECREF(args);                             /* добавить проверку ошибок */

    if (pres != NULL) {
        /* use and decref handler result */
        PyArg_Parse(pres, "s", &cres);
```

[1] Если нужен более практический пример обработчиков обратного вызова Python, посмотрите на систему GUI Tkinter, активно используемую в этой книге. Tkinter использует как расширение, так и встраивание. Его интерфейс *расширения* (объекты графических элементов) используется для регистрации обработчиков обратного вызова Python, которые в дальнейшем выполняются с интерфейсами *встраивания* в ответ на события GUI. За подробностями можно обратиться к исходному коду реализации Tkinter в дистрибутиве Python, хотя чтение затруднено логикой интерфейса библиотеки Tk.

```
        printf("%s\n", cres);
        Py_DECREF(pres);
    }
}

/*****************************************************/
/* 2) модуль расширения python для регистрации обработчиков */
/* python импортирует этот модуль для установки объектов обработчиков */
/*****************************************************/

static PyObject *
Register_Handler(PyObject *self, PyObject *args)
{
    /* save Python callable object */
    Py_XDECREF(Handler);                /* уже вызывался? */
    PyArg_Parse(args, "O", &Handler);   /* один аргумент? */
    Py_XINCREF(Handler);                /* добавить ссылку */
    Py_INCREF(Py_None);                 /* возврат 'None': успех */
    return Py_None;
}

static PyObject *
Trigger_Event(PyObject *self, PyObject *args)
{
    /* моделирование Python события, перехваченного C */
    static count = 0;
    Route_Event("spam", count++);
    Py_INCREF(Py_None);
    return Py_None;
}

static struct PyMethodDef cregister_methods[] = {
    {"setHandler",    Register_Handler},       /* имя, адрес */
    {"triggerEvent",  Trigger_Event},
    {NULL, NULL}
};

void initcregister()                /* вызывается из Python  */
{                                   /* при первом "import cregister" */
    (void) Py_InitModule("cregister", cregister_methods);
}
```

Конечно, этот файл C служит модулем расширения Python, а не самостоятельной программой C со встроенным Python (хотя C мог бы быть верхним уровнем). Для компиляции его в файл динамически загружаемого модуля выполните make-файл примера 20.10 в Linux (и что-нибудь аналогичное на других платформах). Как известно из предшествующей главы, результирующий файл *cregister.so* будет загружен при первом импорте в сценарии Python, если поместить его в каталог, находящийся в пути поиска модулей Python (например, «.»).

Пример 20.10. PP2E\Integrate\Mixed\Regist\makefile.regist

```
#########################################################################
# Создает cregister.so, динамически загружаемый модуль расширения C
# (совместного доступа), который импортируется register.py
#########################################################################

PY    = $(MYPY)
PYINC = -I$(PY)/Include -I$(PY)
```

```
CMODS = cregister.so

all: $(CMODS)

cregister.so: cregister.c
    gcc cregister.c -g $(PYINC) -fpic -shared -o cregister.so

clean:
    rm -f *.pyc $(CMODS)
```

Теперь, когда у нас есть модуль расширения C для регистрации и вызова обработчиков Python, нам нужны лишь какие-нибудь обработчики Python. Модуль Python, показанный в примере 20.11, определяет две функции обработчиков обратного вызова и импортирует модуль расширения C для регистрации обработчиков и генерации событий.

Пример 20.11. PP2E\Integrate\Mixed\Regist\register.py

```
###########################################################################
# зарегистрировать для обработки событий обратными вызовами из C;
# компилировать код C и запускать как 'python register.py'
###########################################################################

#
# эти функции Python вызываются из C;
# обработать событие, возвратить результат
#

def callback1(label, count):
    return 'callback1 => %s number %i' % (label, count)

def callback2(label, count):
    return 'callback2 => ' +  label * count

#
# Python вызывает модуль расширения C
# для регистрации обработчиков, генерации событий
#

import cregister

print '\nTest1:'
cregister.setHandler(callback1)
for i in range(3):
    cregister.triggerEvent()  # моделирование событий, перехватываемых слоем C

print '\nTest2:'
cregister.setHandler(callback2)
for i in range(3):
    cregister.triggerEvent()  # направляет эти события в callback2
```

Вот и все – интеграция обратных вызовов Python/C готова к работе. Для запуска системы выполните сценарий Python; он регистрирует одну функцию, заставляет сгенерироваться три события, затем изменяет обработчик событий и все повторяет:

```
[mark@toy ~/.../PP2E/Integration/Mixed/Regist]$ python register.py

Test1:
callback1 => spam number 0
callback1 => spam number 1
callback1 => spam number 2

Test2:
callback2 => spamspamspam
```

```
callback2 => spamspamspamspam
callback2 => spamspamspamspamspam
```

Этот вывод производит функция маршрутизации событий C, но состоит он из значений, возвращаемых функциями обработчиков в модуле Python. В действительности здесь незаметно происходит бурная деятельность. Когда Python возбуждает событие, между двумя языками происходит такая передача управления:

1. Из Python в функцию маршрутизации событий C
2. Из функции маршрутизации событий C в функцию обработчика Python
3. Обратно в функцию маршрутизации событий C (где отображается вывод)
4. Наконец, обратно в сценарий Python

Таким образом, мы переходим из Python в C, оттуда в Python и еще раз обратно. Попутно управление проходит через интерфейсы расширения и встраивания. Когда работает обработчик обратного вызова Python, активны два уровня Python и один уровень C посередине. К счастью, это действует; API Python реентерабелен, поэтому не нужно беспокоиться по поводу того, что одновременно активны несколько уровней интерпретатора Python. Каждый уровень выполняет свой код и действует независимо.

Использование в C классов Python

В предыдущей главе мы научились использовать классы C++ в Python, создавая для них оболочки с помощью SWIG. А можно ли пойти в обратном направлении – использовать классы Python из других языков? Оказывается, для этого нужно только применить уже продемонстрированные интерфейсы.

Вспомните, что сценарии Python создают объекты экземпляров классов путем *вызова* объектов классов, как если бы они являлись функциями. Чтобы сделать это из C (или C++), нужно просто сделать те же шаги: импортировать класс из модуля (или другого места), создать кортеж аргументов и вызвать класс для создания экземпляра с помощью тех же средств C API, которыми вызываются функции Python. Создав экземпляр, можно получать его атрибуты и методы теми же средствами, с помощью которых получаются глобальные переменные из модуля.

Чтобы продемонстрировать это на практике, в примере 20.12 определен модуль с простым классом Python, которым можно пользоваться в C.

Пример 20.12. PP2E\Integrate\Embed\ApiClients\module.py

```
# вызывать этот класс из C для создания объектов

class klass:
    def method(self, x, y):
        return "brave %s %s" % (x, y)   # запускать из C
```

Проще некуда, но достаточно для иллюстрации основ. Как обычно, этот модуль должен находиться в пути поиска для Python (например, в текущем каталоге или содержащемся в PYTHONPATH), иначе импортировать его при вызове из C не удастся, так же как и в сценарии Python. Теперь вот как можно пользоваться этим классом Python из программы на Python:

```
C:\...\PP2E\Integrate\Embed\ApiClients>python
>>> import module                                # импорт файла
>>> object = module.klass()                      # создать экземпляр класса
>>> result = object.method('sir', 'robin')       # вызвать метод класса
>>> print result
brave sir robin
```

В Python это весьма просто. Те же действия можно выполнить и в C, но для этого потребуется немного больше кода. Файл C в примере 20.13 выполняет эти шаги, организуя вызовы надлежащих средств Python API.

Пример 20.13. PP2E\Integrate\Embed\ApiClients\objects-low.c

```
#include <Python.h>
#include <stdio.h>

main() {
  /* run objects with low-level calls */
  char *arg1="sir", *arg2="robin", *cstr;
  PyObject *pmod, *pclass, *pargs, *pinst, *pmeth, *pres;

  /* instance = module.klass() */
  Py_Initialize();
  pmod   = PyImport_ImportModule("module");          /* загрузить модуль */
  pclass = PyObject_GetAttrString(pmod, "klass");    /* загрузить module.class */
  Py_DECREF(pmod);

  pargs  = Py_BuildValue("()");
  pinst  = PyEval_CallObject(pclass, pargs);         /* вызвать class() */
  Py_DECREF(pclass);
  Py_DECREF(pargs);

  /* result = instance.method(x,y) */
  pmeth  = PyObject_GetAttrString(pinst, "method"); /* загрузить связанный метод */
  Py_DECREF(pinst);
  pargs  = Py_BuildValue("(ss)", arg1, arg2);        /* преобразовать в Python */
  pres   = PyEval_CallObject(pmeth, pargs);          /* вызвать method(x,y) */
  Py_DECREF(pmeth);
  Py_DECREF(pargs);

  PyArg_Parse(pres, "s", ,cstr);                     /* преобразовать с C *.
  printf("%s\n", cstr);
  Py_DECREF(pres);
}
```

Просмотрите этот файл и разберитесь в деталях. Нужно просто понять, как задача решалась бы в Python, а затем вызывать эквивалентные функции C в Python API. Для компиляции этого исходного текста в выполняемую программу C запустите makefile в каталоге, где находится этот файл (он аналогичен уже рассмотренным makefайлам). После компиляции запустите, как любую другую программу C:

```
[mark@toy ~/.../PP2E/Integrate/Embed/ApiClients]$ objects-low
brave sir robin
```

Вывод может разочаровать, но он действительно отражает значения, возвращаемые в C методом класса из файла *module.py*. C пришлось немало потрудиться, чтобы получить эту короткую строку: он импортировал модуль, загрузил класс, создал экземпляр, загрузил и вызвал метод экземпляра, попутно выполняя преобразование данных и управляя счетчиком ссылок. В награду за все труды C может применять показанную в этом файле технику для повторного использования *любых* классов Python.

Конечно, на практике этот пример был бы сложнее. Как уже говорилось, обычно нужно проверять значения, возвращаемые всеми вызовами Python API, чтобы убедиться в их успешном выполнении. Например, вызов импорта модуля в этом коде C может закончиться неудачей, если модуль находится вне пути поиска; если не перехватывать возврат указателя `NULL`, то программа почти наверняка закончится крахом при попытке использовать этот указатель (когда он понадобится). В примере 20.14 код примера 20.13 переписан с добавлением всех проверок ошибок; код велик, но устойчив.

Пример 20.14. PP2E\Integrate\Embed\ApiClients\objects-err-low.c

```
#include <Python.h>
#include <stdio.h>
#define error(msg) do { printf("%s\n", msg); exit(1); } while (1)

main() {
  /* запуск объектов с помощью вызовов низкого уровня с полной проверкой ошибок */
  char *arg1="sir", *arg2="robin", *cstr;
  PyObject *pmod, *pclass, *pargs, *pinst, *pmeth, *pres;

  /* instance = module.klass() */
  Py_Initialize();
  pmod = PyImport_ImportModule("module");           /* загрузить модуль */
  if (pmod == NULL)
      error("Can't load module");

  pclass = PyObject_GetAttrString(pmod, "klass");   /* загрузить module.class */
  Py_DECREF(pmod);
  if (pclass == NULL)
      error("Can't get module.klass");

  pargs = Py_BuildValue("()");
  if (pargs == NULL) {
      Py_DECREF(pclass);
      error("Can't build arguments list");
  }
  pinst = PyEval_CallObject(pclass, pargs);         /* вызвать class() */
  Py_DECREF(pclass);
  Py_DECREF(pargs);
  if (pinst == NULL)
      error("Error calling module.klass()");

  /* result = instance.method(x,y) */
  pmeth  = PyObject_GetAttrString(pinst, "method"); /* загрузить связанный метод */
  Py_DECREF(pinst);
  if (pmeth == NULL)
      error("Can't fetch klass.method");

  pargs = Py_BuildValue("(ss)", arg1, arg2);        /* преобразовать в Python */
  if (pargs == NULL) {
      Py_DECREF(pmeth);
      error("Can't build arguments list");
  }
  pres = PyEval_CallObject(pmeth, pargs);           /* вызвать method(x,y) */
  Py_DECREF(pmeth);
  Py_DECREF(pargs);
  if (pres == NULL)
      error("Error calling klass.method");

  if (!PyArg_Parse(pres, "s", &cstr))               /* преобразовать в C */
     error("Can't convert klass.method result");
  printf("%s\n", cstr);
  Py_DECREF(pres);
}
```

ppembed: API высокого уровня для встраивания

Но не делайте этого. Как можно судить по последнему примеру, в серьезных приложениях код интеграции для встраивания очень быстро становится таким же слож-

ным, как код расширения. На сегодняшний день нет решения для автоматизации задачи встраивания, которое было бы столь же удачным, как применение SWIG для расширения. Поскольку встраивание не налагает таких структурных ограничений, как модули и типы расширений, это в значительно меньшей мере связанная условиями задача; автоматизация одной стратегии встраивания может оказаться совершенно бесполезной для другой.

Однако с помощью некоторой предварительной работы можно все же автоматизировать обычные задачи встраивания, заключая вызовы в API более высокого уровня. Эти API могут справиться с такими вещами, как обнаружение ошибок, счетчики ссылок, преобразование данных и т. д. Один из таких API, *ppembed*, есть на прилагаемом к книге CD. Он просто объединяет существующие в стандартном C API Python средства, предоставляя ряд более простых в обращении вызовов для выполнения программ Python из C.

Выполнение объектов с помощью ppembed

Пример 20.15 показывает *objects-err-low.c*, переработанный для присоединения к программе библиотечных файлов ppembed.

Пример 20.15. PP2E\Integrate\Embed\ApiClients\object-api.c

```
#include <stdio.h>
#include "ppembed.h"

main () {                                   /* с api высокого уровня ppembed */
  int failflag;
  PyObject *pinst;
  char *arg1="sir", *arg2="robin", *cstr;

  failflag = PP_Run_Function("module", "klass", "O", &pinst, "()") ||
             PP_Run_Method(pinst, "method", "s", &cstr, "(ss)", arg1, arg2);

  printf("%s\n", (!failflag) ? cstr : "Can't call objects");
  Py_XDECREF(pinst); free(cstr);
}
```

В этом файле использованы два вызова ppembed (их имена начинаются с «PP»), с помощью которых создается экземпляр класса и вызывается его метод. Так как ppembed занимается проверкой ошибок, счетчиками ссылок, преобразованием данных и т. д., ничего особенного больше делать не приходится. После компоновки этой программы с кодом библиотеки ppembed она работает как первоначальная, но читать, писать и отлаживать ее значительно проще:

```
[mark@toy ~/.../PP2E/Integrate/Embed/ApiClients]$ objects-api
brave sir robin
```

Выполнение строк кода с помощью ppembed

API ppembed предоставляет вызовы высокого уровня для большинства приемов встраивания, рассмотренных в этой главе. Например, программа C в примере 20.16 выполняет строки кода, которые заставляют модуль `string` записать простой текст заглавными буквами.

Пример 20.16. PP2E\Integrate\Embed\ApiClients\codestring-low.c

```
#include <Python.h>          /* определения стандартного API */
void error(char *msg) { printf("%s\n", msg); exit(1); }
```

```
main() {
    /* выполнение строк с помощью вызовов низкого уровня */
    char *cstr;
    PyObject *pstr, *pmod, *pdict;              /* с проверкой ошибок */
    Py_Initialize();

    /* result = string.upper('spam') + '!' */
    pmod = PyImport_ImportModule("string");     /* загрузить модуль */
    if (pmod == NULL)                           /* для пространства имен */
        error("Can't import module");

    pdict = PyModule_GetDict(pmod);             /* string.__dict_*/
    Py_DECREF(pmod);
    if (pdict == NULL)
        error("Can't get module dict");

    pstr = PyRun_String("upper('spam') + '!'", Py_eval_input, pdict, pdict);
    if (pstr == NULL)
        error("Error while running string");

    /* преобразовать результат в C */
    if (!PyArg_Parse(pstr, "s", &cstr))
        error("Bad result type");
    printf("%s\n", cstr);
    Py_DECREF(pstr);        /* освободить экспортированные объекты, не pdict */
}
```

Программа C в этом файле содержит политически корректные проверки ошибок после каждого вызова API. При запуске она выводит результат выполнения вызова преобразования в верхний регистр в пространстве имен модуля Python `string`:

```
[mark@toy ~/.../PP2E/Integrate/Embed/ApiClients]$ codestring-low
SPAM!
```

Такую интеграцию можно осуществлять, вызывая функции Python API непосредственно, но делать это необязательно. С помощью API встраивания высокого уровня типа ppembed задача может быть заметно упрощена, как это показано в примере 20.17.

Пример 20.17. PP2E\Integrate\Embed\ApiClients\codestring-api.c

```
#include "ppembed.h"
#include <stdio.h>
                                                    /* с api высокого уровня ppembed */
main() {
    char *cstr;
    int err = PP_Run_Codestr(
                    PP_EXPRESSION,                  /* выражение или оператор?  */
                    "upper('spam') + '!'", "string",    /* код, модуль */
                    "s", &cstr);                    /* результат выражения */
    printf("%s\n", (!err) ? cstr : "Can't run string");  /* и free(cstr) */
}
```

После компоновки с кодом библиотеки ppembed эта версия дает такой же результат, как предыдущая. Как и большинство API верхнего уровня, ppembed исходит из некоторых предположений относительно способа применения, которые не всегда выполняются; однако если предположения соблюдены для решаемой задачи встраивания, то такие вызовы-оболочки значительно разгружают программы, которым требуется выполнение встроенного кода Python.

Выполнение настраиваемых проверок

Встроенный код Python может выполнять и полезную работу. Например, программа C в примере 20.18 вызывает функции ppembed для выполнения строки кода Python, загружаемой из файла, которая выполняет проверку действительности инвентаризационных данных. Для экономии места я не стану перечислять все компоненты, используемые в этом примере (хотя можно найти их на прилагаемом к книге CD). Тем не менее, этот файл показывает те части встраивания, которые относятся к этой главе: он устанавливает переменные в пространстве имен кода Python, служащие в качестве входных данных, выполняет код Python, а затем загружает имена из пространства имен кода в качестве результатов.[1]

Пример 20.18. PP2E\Integrate\Embed\Inventory\order-string.c

```
/* выполнение проверки в строках кода */

#include <ppembed.h>
#include <stdio.h>
#include <string.h>
#include "ordersfile.h"

run_user_validation()
{                                       /* Python инициализируется автоматически */
    int i, status, nbytes;              /* предостережение: статус должен проверяться всюду */
    char script[4096];                  /* предостережение: нужно выделить память достаточно
                                                   большому блоку */
    char *errors, *warnings;
    FILE *file;

    file = fopen("validate1.py", "r");          /* настраиваемые проверки */
    nbytes = fread(script, 1, 4096, file);      /* загрузить текст файла Python */
    script[nbytes] = '\0';

    status = PP_Make_Dummy_Module("orders");  /* собственное пространство имен приложения */
    for (i=0; i < numorders; i++) {           /* подобно созданию нового словаря */
        printf("\n%d (%d, %d, '%s')\n",
            i, orders[i].product, orders[i].quantity, orders[i].buyer);

        PP_Set_Global("orders", "PRODUCT",  "i", orders[i].product);   /* int */
        PP_Set_Global("orders", "QUANTITY", "i", orders[i].quantity);  /* int */
        PP_Set_Global("orders", "BUYER",    "s", orders[i].buyer);     /* str */

        status = PP_Run_Codestr(PP_STATEMENT, script, "orders", "", NULL);
        if (status == -1) {
            printf("Python error during validation.\n");
            PyErr_Print();  /* show traceback */
            continue;
        }

        PP_Get_Global("orders", "ERRORS",   "s", &errors);     /* может разбивать */
        PP_Get_Global("orders", "WARNINGS", "s", &warnings);   /* по пробелам */

        printf("errors:   %s\n", strlen(errors)? errors : "none");
        printf("warnings: %s\n", strlen(warnings)? warnings : "none");
```

[1] Это более или менее тот вид структуры, которая используется при встраивании Python в файлы HTML в расширении Active Scripting, за исключением того, что устанавливаемые здесь глобальные переменные (например, PRODUCT) становятся именами, предустановленными для объектов веб-броузера, а код извлекается из веб-страницы, а не из текстового файла с известным именем. Смотрите главу 15.

```
        free(errors); free(warnings);
        PP_Run_Function("inventory", "print_files", "", NULL, "()");
    }
}

main(int argc, char **argv)        /* С наверху, Python встроен, но Python */
{                                  /* может тоже использовать расширения С */
    run_user_validation();         /* во встроенном коде не требуется sys.argv */
}
```

Здесь нужно отметить ряд вещей. Прежде всего, на практике эта программа может получать имя файла с кодом Python или путь к нему из настраиваемых переменных окружения; здесь он загружается из текущего каталога. Во-вторых, можно было написать эту программу с помощью простых вызовов API вместо ppembed, но все вызовы «PP» превратились бы в блоки более сложного кода. В настоящем виде для создания программы можно компилировать и компоновать этот файл с помощью файлов библиотек Python и ppembed. Код Python, выполняемый результирующей программой С, находится в примере 20.19; он использует предустановленные глобальные переменные и должен присваивать им значения для отправки результирующих строк обратно в С.

Пример 20.19. PP2E\Integrate\Embed\Inventory\validate1.py

```
# встроенный код проверки, выполняемый из С
# входные переменные:  PRODUCT, QUANTITY, BUYER
# выходные переменные: ERRORS, WARNINGS

import string              # встроенному коду доступны все средства Python
import inventory           # плюс расширения С, модули Python, классы...
msgs, errs = [], []        # списки предупреждений, сообщений об ошибках

def validate_order():
    if PRODUCT not in inventory.skus():  # эту функцию можно было импортировать
        errs.append('bad-product')       # также из определенного пользователем
                                         # модуля
    elif QUANTITY > inventory.stock(PRODUCT):
        errs.append('check-quantity')
    else:
        inventory.reduce(PRODUCT, QUANTITY)
        if inventory.stock(PRODUCT) / QUANTITY < 2:
            msgs.append('reorder-soon:' + `PRODUCT`)

first, last = BUYER[0], BUYER[1:]         # код можно менять на месте:
if first not in string.uppercase:         # этот файл выполняется как одна длинная
    errs.append('buyer-name:' + first)    # строка кода, с входными и выходными
if BUYER not in inventory.buyers():       # переменными, используемыми приложением С
    msgs.append('new-buyer-added')
    inventory.add_buyer(BUYER)
validate_order()

ERRORS   = string.join(errs)      # вставить пробел между сообщениями
WARNINGS = string.join(msgs)      # выдавать как строки: "" == none
```

Не мучайтесь над деталями этого кода; некоторые используемые в нем компоненты тоже не показаны (полная реализация есть на прилагаемом CD). Однако нужно заметить, что этот файл кода может содержать любой код Python – в нем можно определять функции и классы, использовать сокеты и потоки и т. д. Встраивая Python, вы даром получаете полнофункциональный язык расширений. Может быть, еще важнее то, что поскольку этот файл представляет собой код Python, в него можно вносить лю-

бые изменения, а перекомпиляции программы C при этом не требуется. Такая гибкость особенно полезна, когда система уже поставлена и установлена.

Как говорилось ранее, есть много способов организации встроенного кода Python. Например, аналогичной гибкости можно добиться, передавая действия *функциям* Python, загружаемым из файлов *модулей*, как показано в примере 20.20.

Пример 20.20. PP2E\Integrate\Embed\Inventory\order-func.c

```
/* выполнение проверок функциями встраиваемого модуля */

#include <ppembed.h>
#include <stdio.h>
#include <string.h>
#include "ordersfile.h"

run_user_validation() {
    int i, status;             /* статус должен проверяться всюду */
    char *errors, *warnings;  /* здесь нет файла/строки или пространства имен */
    PyObject *results;

    for (i=0; i < numorders; i++) {
        printf("\n%d (%d, %d, '%s')\n",
            i, orders[i].product, orders[i].quantity, orders[i].buyer);

        status = PP_Run_Function(                /* validate2.validate(p,q,b) */
                        "validate2", "validate",
                        "O",           &results,
                        "(iis)",       orders[i].product,
                                       orders[i].quantity, orders[i].buyer);
        if (status == -1) {
            printf("Python error during validation.\n");
            PyErr_Print();  /* show traceback */
            continue;
        }
        PyArg_Parse(results, "(ss)", &warnings, &errors);
        printf("errors:   %s\n", strlen(errors)? errors : "none");
        printf("warnings: %s\n", strlen(warnings)? warnings : "none");
        Py_DECREF(results);  /* ok to free strings */
        PP_Run_Function("inventory", "print_files", "", NULL, "()");
    }
}

main(int argc, char **argv) {
    run_user_validation();
}
```

Разница здесь в том, что файл с кодом Python (показанный в примере 20.21) импортируется и потому должен находиться в пути поиска модулей Python. Предполагается также, что он содержит функции, а не просто ряд операторов. Строки можно располагать всюду – в файлах, базах данных, веб-страницах и т. д., и кодировать их конечному пользователю, возможно, проще. Но надо думать, что дополнительные требования функций в модулях не являются чрезмерными, зато функции предоставляют естественную модель связи в виде аргументов и возвращаемых значений.

Пример 20.21. PP2E\Integrate\Embed\Inventory\validate2.py

```
# встроенный код проверки, выполняется из C
# вход = аргументы, выход = кортеж возвращаемых значений

import string
```

```
import inventory

def validate(product, quantity, buyer):           # функция, вызываемая по имени
    msgs, errs = [], []                           # через строки имен модуль/функция
    first, last = buyer[0], buyer[1:]
    if first not in string.uppercase:
        errs.append('buyer-name:' + first)
    if buyer not in inventory.buyers():
        msgs.append('new-buyer-added')
        inventory.add_buyer(buyer)
    validate_order(product, quantity, errs, msgs)  # изменяемый список аргументв
    return string.join(msgs), string.join(errs)    # использовать формат "(ss)"

def validate_order(product, quantity, errs, msgs):
    if product not in inventory.skus():
        errs.append('bad-product')
    elif quantity > inventory.stock(product):
        errs.append('check-quantity')
    else:
        inventory.reduce(product, quantity)
        if inventory.stock(product) / quantity < 2:
            msgs.append('reorder-soon:' + `product`)
```

Реализация ppembed

API ppembed сначала возник как пример в первом издании этой книги. Со временем он стал использоваться в реальных системах и слишком разросся, чтобы его можно было представить здесь целиком. Например, ppembed поддерживает также отладку встраиваемого кода (переадресуя его в модуль отладчика `pdb`), динамическую перегрузку модулей, содержащих встраиваемый код, и другие функции, слишком сложные для полноценного показа здесь.

Но при желании изучить еще один пример применения вызовов Python для встраивания можно взять полный исходный код ppembed и make-файл с прилагаемого CD из каталога:

PP2E\Integration\Embed\HighLevelApi

В качестве примера того, какие средства можно создавать для упрощения встраивания, в примере 20.22 приводится заголовочный файл API ppembed. Его код можно по желанию изучать, использовать, копировать и совершенствовать. А можно просто написать собственный API. Главное, что нужно понять из этого раздела, это то, что встраивание программ оказывается сложным, только если придерживаться API исполнения Python в том виде, как он поставляется. Добавляя вспомогательные функции, как в ppembed, можно сделать встраивание значительно проще. Кроме того, программы C становятся независимыми от изменений в ядре Python C; в идеале, если изменится Python, нужно будет поменять только API.

Посмотрите также файл *abstract.h* в каталоге Python включаемых файлов, если ищете интерфейсы высокого уровня. В этом файле содержатся общие вызовы операций над типами, которые облегчают такие действия, как создание, наполнение содержанием, доступ по индексу, осуществление срезов и конкатенацию объектов Python, к которым обращаются по указателям из C. Посмотрите также соответствующий файл реализации *abstract.c* и реализации встроенных модулей и типов Python в дистрибутиве исходного кода Python, где есть другие примеры доступа к объектам на низком уровне. Получив указатель на объект Python в C, можно осуществлять любые специфические для типов действия с входными и выходными данными Python.

Пример 20.22. PP2E\Integrate\Embed\HighLevelApi\ppembed.h

```
/*************************************************************************
 * PPEMBED, VERSION 2.0
 * УСОВЕРШЕНСТВОВАННЫЙ ИНТЕРФЕЙС ВЫЗОВА ВСТРОЕННОГО PYTHON
 *
 * Создает оболочки функций API встраивания Python, облегчающие их применение.
 * В большинстве утилит предполагается уточнение вызова охватывающим модулем
 * (пространством имен). Модуль может задаваться именем файла или быть фиктивным модулем,
 * создаваемым для предоставления пространства имен строкам вне файла. Эти функции
 * автоматизируют отладку, (пере)загрузку модулей, преобразования при вводе/выводе и т. д.
 *
 * Python автоматически инициализируется при первом обращении к API. В преобразованиях
 * ввода/вывода используются стандартные коды форматов преобразований Python (описаны
 * в руководстве по C API). На ошибки указывает результат в виде целого -1 или указателя NULL.
 * Экспортируемые имена имеют префикс PP для предотвращения конфликтов; имена встроенного API
 * Python используют префиксы Py  (к сожалению, в C нет эквивалента "import", только "from*").
 * Кроме того, код varargs может оказаться непереносимым в некоторые компиляторы C;
 * об обеспечении переносимости читайте в книге или файле 'vararg.txt'
 * либо ищите строку STDARG в файлах исходного кода Python.
 *
 * Новое в этой версии/издании: имена содержат префикс PP, файлы
 * переименованы, компиляция в один файл .a, исправлена ошибка pdb retval
 * для строк, результаты char*, возвращаемые кодом преобразования "s", указывают
 * на новые массивы char, которые вызвавший должен free(), когда в них пропадает
 * наобходимость (в предыдущей версии это служило причиной ошибок).
 * Добавлены новые интерфейсы API для получения info исключительных ситуаций,
 * прекомпиляции строк кода в байт-код и вызова простых объектов.
 *
 * Полностью поддерживается импорт пакетов модулей Python 1.5: имена модулей
 * в этом API могут иметь вид "package.package.[...].module", где
 * Python отображает имена пакетов во вложенные каталоги иерархии
 * файловой системы; все каталоги пакетов содержат файлы __init__.py,
 * а самый левый находится в каталоге, указанном в PYTHONPATH. Функция
 * динамической перегрузки этого API действует для модулей в пакетах;
 * Python хранит полное имя пути в словаре sys.modules.
 *
 * Предупреждение: не поддерживаются такие сложные функции, как потоки и
 * режим ограниченного выполнения, но их можно ввести с помощью дополнительных
 * вызовов Python API вне данного API (см. в руководстве по Python/C API
 * о вызовах потоков для уровня C; см. в библиотечном руководстве по
 * модулям rexec и bastion относительно ограниченного режима). С потоками
 * можно справиться с помощью потоков C и отдельных пространств имен Python
 * для каждого сегмента кода Python, или запускать потоки Python
 * из кода Python, выполняемого из C (см. модуль Python thread).
 *
 * Учтите, что Python может перегружать только модули Python, но не расширения C,
 * но можно выставлять флаг динамической перегрузки, даже при обращении
 * к динамически загружаемым модулям расширений C - в 1.5.2 Python
 * устанавливает для модулей расширения C первоначальное состояние атрибутов
 * при перезагрузке, но на деле не перегружает файлы расширений C.
 *************************************************************************/

#ifndef PPEMBED_H
#define PPEMBED_H

#ifdef __cplusplus
extern "C" {            /* Библиотека C, но может вызываться из C++ */
#endif
```

```
#include <stdio.h>
#include <Python.h>

extern int PP_RELOAD;    /* 1=перегрузить модули ру при обращении к атрибутам */
extern int PP_DEBUG;     /* 1=запустить отладчик при выполнении
                                                    строки/функции/члена */

typedef enum {
     PP_EXPRESSION,      /* тип строки кода */
     PP_STATEMENT        /* выражения и операторы различаются */
} PPStringModes;

/*************************************************/
/*  ppembed-modules.c: загрузка, обращение к объектам модуля */
/*************************************************/

extern char     *PP_Init(char *modname);
extern int       PP_Make_Dummy_Module(char *modname);
extern PyObject *PP_Load_Module(char *modname);
extern PyObject *PP_Load_Attribute(char *modname, char *attrname);
extern int       PP_Run_Command_Line(char *prompt);

/*********************************************************/
/*  ppembed-globals.c: чтение, запись переменных уровня модуля  */
/*********************************************************/

extern int
    PP_Convert_Result(PyObject *presult, char *resFormat, void *resTarget);

extern int
    PP_Get_Global(char *modname, char *varname, char *resfmt, void *cresult);

extern int
    PP_Set_Global(char *modname, char *varname, char *valfmt, ... /*val*/);

/*************************************************/
/*  ppembed-strings.c: выполнение строк кода Python */
/*************************************************/

extern int                                      /* выполнить строку C с кодом */
    PP_Run_Codestr(PPStringModes mode,          /* выражение или оператор?  */
                   char *code,   char *modname,   /* строка кода, пространство имен модуля */
                   char *resfmt, void *cresult);  /* тип результата, цель */

extern PyObject*
    PP_Debug_Codestr(PPStringModes mode,        /* выполнить строку в pdb */
                     char *codestring, PyObject *moddict);

extern PyObject *
    PP_Compile_Codestr(PPStringModes mode,
                       char *codestr);          /* компиляция в байт-код */

extern int
    PP_Run_Bytecode(PyObject *codeobj,          /* выполнить объект байт-кода */
                    char     *modname,
                    char     *resfmt, void *restarget);

extern PyObject *                               /* выполнить байт-коды в pdb */
    PP_Debug_Bytecode(PyObject *codeobject, PyObject *moddict);

/*************************************************/
/*  ppembed-callables.c: вызов функций, классов и т. д. */
/*************************************************/
```

```
extern int                                                  /* mod.func(args) */
    PP_Run_Function(char *modname, char *funcname,          /* func|classname */
                    char *resfmt,  void *cresult,           /* цель результата */
                    char *argfmt,  ... /* arg, arg... */ ); /* входные аргументы */

extern PyObject*
    PP_Debug_Function(PyObject *func, PyObject *args);      /* вызов func в pdb */

extern int
    PP_Run_Known_Callable(PyObject *object,                 /* func|class|method */
                          char *resfmt, void *restarget,    /* пропустить загрузку модуля */
                          char *argfmt, ... /* arg,.. */ );


/*************************************************************/
/*  ppembed-attributes.c: выполнение методов объекта, доступ к членам*/
/*************************************************************/

extern int
    PP_Run_Method(PyObject *pobject, char *method,              /* использует Debug_Function */
                  char *resfmt,  void *cresult,                 /* выход */
                  char *argfmt,  ... /* arg, arg... */ );      /* вход */

extern int
    PP_Get_Member(PyObject *pobject, char *attrname,
                  char *resfmt,  void *cresult);                /* выход */

extern int
    PP_Set_Member(PyObject *pobject, char *attrname,
                  char *valfmt,  ... /* val, val... */ );      /* вход */


/*******************************************************/
/*  ppembed-errors.c: получить данные исключения после ошибки api */
/*******************************************************/

extern void PP_Fetch_Error_Text();    /* получить (и сбросить) исключение */

extern char PP_last_error_type[];     /* текст имени исключения */
extern char PP_last_error_info[];     /* текст данных исключения  */
extern char PP_last_error_trace[];    /* текст трассировки исключения */

extern PyObject *PP_last_traceback;   /* сохраненный объект трассировки исключения */


#ifdef __cplusplus
}
#endif

#endif (!PPEMBED_H)
```

Другие примеры интеграции, имеющиеся на CD

Когда я писал эту главу, отведенное для нее место закончилось раньше, чем закончились примеры. Кроме примера API ppembed из последнего раздела на прилагаемом к книге CD можно найти ряд других примеров интеграции Python/C для самостоятельного изучения:

PP2E\Integration\Embed\Inventory

Полная реализация примеров подтверждения, приведенных выше. В этом конкретном примере API ppembed используется для выполнения проверок порядка встраиваемым Python, в виде как встраиваемых строк кода, так и функций, загру-

жаемых из модулей. Список реализован как с полками и файлами сериализации для постоянного хранения данных, так и без них.

PP2E\Integration\Mixed\Exports

Средство экспортирования переменных C для использования во встраиваемых программах Python.

PP2E\Integration\Embed\TestApi

Простая программа тестирования ppembed, с путями импорта пакетов для идентификации модулей и без них.

Некоторые из них являются большими примерами C, которые, вероятно, лучше изучать, чем приводить в листинге.

Другие темы интеграции

В данной книге термин *интеграция* в основном означал соединение Python с компонентами, написанными на C или C++ (либо другими совместимыми с C языками), в режимах расширения или встраивания. Но с более широкой точки зрения интеграция также включает в себя любые другие технологии, позволяющие образовывать из компонентов Python крупные системы. В данном последнем разделе кратко рассматривается ряд технологий интеграции помимо средств C API, рассматриваемых в этой части книги.

Интеграция с помощью JPython (или Jython)

С JPython мы познакомились в главе 15, но стоит еще раз упомянуть его в контексте интеграции в целом. Как мы уже видели, JPython поддерживает два вида интеграции:

- JPython использует *API отражения* Java, чтобы позволить программам Python автоматически обращаться к библиотекам классов Java (расширение). API отражения Java предоставляет информацию о типах Java на этапе исполнения и служит той же цели, что и связующий код, который мы генерировали в этой части книги для подключения библиотек C к Python. Однако в JPython эта информация о типах в момент выполнения позволяет в значительной мере автоматизировать разрешение вызовов Java в сценариях Python – не нужно писать или генерировать связующий код.
- JPython также предоставляет API класса Java `PythonInterpreter`, позволяющий программам Java выполнять код Python в пространстве имен (встраивание), подобно средствам C API, использовавшимся для запуска строк кода Python из программ C. Кроме того, поскольку JPython реализует все объекты Python как экземпляры класса Java `PyObject`, слою Java, который содержит встроенный код Python, просто обрабатывать объекты Python.

Иными словами, JPython позволяет Python как расширяться, так и встраиваться в Java, подобно стратегиям интеграции с C, рассмотренным в этой части книги. С вводом системы JPython можно интегрировать Python с любой C-совместимой программой с помощью средств C API, как и с любой Java-совместимой программой с помощью JPython.

Хотя JPython предоставляет удивительно гладкую модель интеграции, код Python в реализации JPython выполняется медленнее, а его опора на библиотеки классов Java и среду выполнения вводит зависимость от Java, которая может быть важна в некоторых сценариях разработки. Подробнее о JPython смотрите в главе 15; полностью вопрос освещен в документации, размещенной в сети на *http://www.jpython.org* (и доступной в пакете JPython на прилагаемом к книге CD).

Интеграция в Windows с помощью COM

Вкратце мы говорили о поддержке в Python объектной модели COM под Windows при рассмотрении Active Scripting в главе 15, но в действительности это общее средство интеграции, полезное не только в Интернете.

Вспомните, что COM определяет стандартную и независимую от языка объектную модель, с помощью которой компоненты, написанные на разных языках, могут интегрироваться и поддерживать связь. Средства пакета расширения Python `win32all` для Windows позволяют программам Python реализовывать серверы и клиенты в модели интерфейса COM.

По существу, COM предоставляет мощный способ интеграции программ Python с программами, написанными на других поддерживающих COM языках, например, Visual Basic, Delphi, Visual C++, PowerBuilder, и даже другими программами Python. Сценарии Python могут также с помощью вызовов COM управлять такими популярными приложениями Microsoft, как Word и Excel, так как эти системы регистрируют собственные интерфейсы объектов COM. Кроме того, новая реализация Python (с экспериментальным названием Python.NET) для технологии Microsoft C#/.NET, о которой говорилось в главе 15, предоставляет еще один способ соединения Python с другими компонентами Windows.

Недостатком COM является необходимость в уровне косвенной адресации и возможность применения в настоящее время только в Windows. Из-за этого данная модель уступает в скорости и переносимости некоторым схемам интеграции на низком уровне, которые изучались в этой части книги (прямой компоновкой, внутрипроцессной и прямым вызовам между Python и компонентам C-совместимых языков). COM также считается крупной системой для серьезного использования, и дальнейшие касающиеся его подробности далеко выходят за рамки данной книги.

Дополнительную информацию о поддержке COM и других расширений Windows можно найти в главе 15 этой книги и в книге O'Reilly «Python Programming on Win32». В указанной книге также значительно подробнее описано использование компиляторов Windows для осуществления интеграции Python и C, чем это можно было бы сделать в данной книге. Например, там показано, как с помощью средств Visual C++ компилировать и компоновать код интегрирующего слоя для Python и C/C++. Базовый код C для расширения и встраивания низкого уровня в Windows такой же, как показанный в этой книге, но детали компиляции и компоновки различаются.

Интеграция CORBA

Имеется также существенная поддержка, частично open source, использования Python в контексте основанных на CORBA приложений. *CORBA* (Common Object Request Broker – стандартная архитектура брокера объектных запросов) представляет собой независимый от языка способ построения распределенных систем из компонентов, связанных между собой архитектурой объектной модели. По существу, она представляет собой еще один способ интеграции компонентов Python в большие системы.

Поддержка CORBA в Python включает в себя находящиеся в общественном владении системы *ILU* (от Xerox) и *fnorb* (смотрите *http://www.python.org*). На момент написания этой книги *OMG* (Object Management Group, отвечающая за направление развития CORBA) также руководит проектом по избранию Python в качестве стандартного языка сценариев для основанных на CORBA систем. Произойдет это в конце концов или нет, но Python служит идеальным языком для программирования распределенных объектов и используется в таком качестве многими компаниями по всему миру.

Как и COM, CORBA является большой системой – слишком большой, чтобы можно было даже поверхностно коснуться ее в этой книге. Дополнительные сведения ищите на сайте Python в материалах, относящихся к CORBA.

Интеграция против оптимизации

При наличии такого количества вариантов интеграции выбор одного из них может быть сложен. Когда, например, следует выбрать что-нибудь вроде COM вместо написания модуля расширения C? Как обычно, это прежде всего зависит от того, почему вам нужно соединять внешние компоненты с программами Python.

В основе своей такие среды, как JPython, COM и CORBA, позволяют сценариям Python усиливать имеющиеся библиотеки программных компонентов и успешно решать такие задачи, как повторное использование кода и интеграция. Однако они почти никак не влияют на оптимизацию: интегрированные компоненты не обязательно быстрее, чем эквиваленты Python.

С другой стороны, модули и типы расширений Python, написанные на компилируемых языках типа C, выполняют две роли: они также могут использоваться для интеграции существующих компонентов, но и часто оказываются более удачным подходом при повышении производительности системы.

Роль среды

Рассмотрим картину в целом. Такие среды, как COM и CORBA, можно, вероятно, рассматривать как альтернативы технологиям интеграции Python/C, с которыми мы познакомились в этой части книги. Например, оформление логики Python в виде *COM-сервера* позволяет осуществлять нечто родственное *встраиванию* – многие языки (включая C) могут обращаться к нему с помощью интерфейсов клиентов COM, с которыми мы познакомились в главе 15. И, как мы видели ранее, JPython позволяет Java встраивать и выполнять код и объекты Python через интерфейс класса Java.

Кроме того, среда позволяет сценариям Python использовать существующие библиотеки компонентов: стандартные библиотеки классов Java в JPython, библиотеки серверов COM в Windows и т. д. В таком качестве внешние библиотеки, открываемые такими средами, более или менее аналогичны модулям *расширения* Python. Например, сценарии Python, использующие интерфейсы *клиентов* COM для доступа к внешним объектам, действуют подобно импортерам модулей расширения C (хотя и через слой косвенной адресации COM).

Роль модулей расширения

API C для Python во многом предназначен для использования в тех же ролях. Как мы видели, модули расширения C могут также выступать в качестве средств повторного использования кода и интеграции – существующие библиотеки C и C++ легко подключаются к Python с помощью SWIG. В большинстве случаев просто генерируется и импортируется связующий код, создаваемый SWIG, благодаря чему почти все существующие компилированные библиотеки могут использоваться в сценариях Python[1].

[1] На самом деле подключать библиотеки с помощью SWIG настолько просто, что расширения обычно лучше всего сначала кодировать как библиотеки C/C++, а позднее делать для них оболочки для использования с Python с помощью SWIG. Добавление слоя COM к существующей библиотеке C может оказаться не столь простым, но во всяком случае менее переносимым – COM в настоящее время является технологией только для Windows.

Кроме того, API встраивания Python позволяет другим языкам выполнять код Python подобно интерфейсам клиента в COM.

Однако одной из главных причин написания модулей расширения C является *оптимизация*: ключевые участки приложений Python можно реализовать или переписать в виде модулей расширения C или C++, чтобы ускорить работу системы в целом (как в примерах стеков из предыдущей главы). Перенос таких компонентов в компилированные модули расширений не только повышает производительность систем, но и является совершенно гладким – интерфейсы модулей в Python выглядят одинаково независимо от языка программирования, на котором реализован модуль.

Выбор технологии интеграции

Напротив, JPython, COM и CORBA непосредственно не решают задач оптимизации вообще; они служат только интеграции. Например, JPython позволяет сценариям Python автоматически получить доступ к библиотекам Java, но в целом обязывает расширения, отличные от Python, кодировать на Java – языке, который сам обычно интерпретируется и не отличается скоростью. COM и CORBA сосредоточены на интерфейсах между компонентами и не определяют язык реализации компонентов. Экспортируя класс Python как COM-сервер, например, можно обеспечить широкое повторное использование его средств в Windows, но к повышению производительности это не имеет отношения.

Из-за своей различной направленности среды не вполне могут заменить менее косвенно действующие модули и типы расширения Python/C, которые мы изучали в последних двух главах, и более косвенны (а потому медленнее), чем API C для встраивания Python. Можно смешивать разные компоненты, но комбинации редко оказываются лучше своих составных частей. Например, хотя библиотеки C можно добавлять в Java с помощью интерфейса машинно-зависимых вызовов, такое занятие небезопасно и непросто. И хотя библиотеки C можно также заключать в COM-серверы, чтобы сделать их видимыми сценариям Python в Windows, в конечном результате, вероятно, получится более медленное и не менее сложное решение, чем при более непосредственном связывании модулей расширения Python.

Как видите, в области интеграции есть множество вариантов. Лучший совет, который я могу дать на прощание, это учитывать, что разные средства предназначены для решения разных задач. Модули и типы расширения C идеальны для оптимизации систем и интеграции библиотек, но структуры предлагают другие способы интеграции компонентов – JPython для соединения со средствами Java, COM для повторного использования и публикации объектов в Windows и т. д. Как всегда, ваш опыт может оказаться иным.

VI

Финал

Последнюю часть книги составляют:

- Глава 21 «Заключение: Python и цикл разработки». Обсуждаются роли и область применения Python.
- Приложение A «Последние изменения в Python». В этом приложении представлены изменения, произошедшие в Python после выхода первого издания книги.
- Приложение B «Прагматика». Здесь приведены детали установки и использования Python.
- Приложение C «Python и C++». В этом приложении, предназначенном для разработчиков на C++, сопоставлены модели классов Python и C++.

Обратите внимание на отсутствие приложений справочного характера. За дополнительными справочными материалами обращайтесь к стандартным руководствам Python, имеющимся на прилагаемом к книге компакт-диске, или коммерческим печатным справочникам, например «*Python Pocket Reference*» издания O'Reilly. Дополнительный материал по базовому языку *Python* есть в «*Learning Python*» издания O'Reilly. А за помощью по другим относящимся к Python темам обращайтесь к ресурсам, указанным в конце приложения B, и на официальный сайт Python *http://www.python.org*.

21

Заключение: Python и цикл разработки

«Книга заканчивается, пора уже и о смысле жизни»

Или, во всяком случае, о смысле Python. Во введении к этой книге я обещал, что мы вернемся к вопросу о задачах, решаемых на Python, после того как посмотрим на его применение на практике. Поэтому в завершение приведу некоторые совершенно субъективные замечания и обобщенные выводы, касающиеся этого языка.

Как говорилось в первой главе, Python нацелен на *производительность* и *интеграцию*. Надеюсь, что эта книга продемонстрировала некоторые выгоды, приносимые такой направленностью. Обратимся в этом заключении от деревьев обратно к лесу – снова рассмотрим выполняемые Python роли, ведя разговор более конкретно. В частности, использование Python в качестве инструмента создания прототипов может оказать глубокое влияние на цикл разработки.

«Как-то мы неправильно программируем компьютеры»

Это одна из наиболее затасканных фраз в нашей отрасли. Все же, если на некоторое время задуматься о картине в целом, большинство из нас, видимо, согласится, что что-то мы «недоделали». За несколько последних десятилетий индустрия программного обеспечения достигла значительного прогресса в совершенствовании процесса разработки (кто-нибудь помнит еще ввод данных с перфокарт?). Но в то же время стоимость разработки потенциально полезных компьютерных программ часто настолько высока, что они становятся невыгодными.

Кроме того, при создании систем с помощью современных инструментов и парадигм часто наблюдается значительное отставание от графика работ. Разработка программного обеспечения по-прежнему не поддается количественным методам, применяемым в других областях конструирования. В мире программирования не столь уж редко для оценки времени выполнения нового проекта берут лучшую оценку и умножают на коэффициент два или три, чтобы учесть непредвиденные затраты при разработке. Очевидно, что такая ситуация не удовлетворяет менеджеров, разработчиков и конечных пользователей.

«Фактор Гиллигана»

Было предположено (в шутку, конечно), что если бы у разработчиков программного обеспечения был святой покровитель, этой чести не удостоился бы никто, кроме Гиллигана, персонажа весьма популярного американского телешоу 1960-х «Остров Гиллигана». Гиллиган – загадочный, обутый в тапочки первый помощник капитана, ви-

новатый, по общему мнению людей, оказавшихся на острове, в происшедшем кораблекрушении.

В самом деле ситуация с Галлиганом кажется до странности знакомой. Выброшенные на необитаемый остров и располагающие лишь самыми скромными из достижений современной технологии, Галлиган и его товарищи вынуждены бороться за выживание, используя лишь то, что дает им природа. В одном эпизоде за другим мы видим, как Профессор изобретает крайне замысловатые средства, чтобы улучшить их жизнь на этом заброшенном острове, лишь для того, чтобы на стадии реализации вечно неловкий Гиллиган все испортил.

Но ясно, что виноват в этом не несчастный Гиллиган. Можно ли рассчитывать, что с помощью элементарных технологий, доступных в таких условиях, будут реализованы проекты таких сложных устройств, как бытовая техника и устройства связи? У него просто не было необходимых средств. Не исключено, что инженерные способности Гиллигана были на высочайшем уровне, но что можно собрать из бананов и кокосов?

И патологически, раз за разом, Гиллиган кончает тем, что непреднамеренно саботирует лучшие планы Профессора, неправильно используя и в конечном счете уничтожая его изобретения. Если бы он крутил педали своего самодельного неподвижного велосипеда быстрее (как его убеждали), все было бы хорошо. Но в конце неизбежно кокосы летели в воздух, а обломки пальмовых ветвей падали ему на голову, и бедного Гиллигана в очередной раз ругали за провал технологии.

Как ни драматичен этот образ, некоторые наблюдатели считают его поразительной метафорой программной индустрии. Как и Гиллигана, нас, разработчиков программ, часто вынуждают решать задачи с помощью явно негодных средств. Как и у Гиллигана, наши намерения правильны, но сдерживает технология. И, как несчастный Гиллиган, мы неизбежно навлекаем на себя гнев руководства, когда наши проекты завершаются с опозданием. С бананами и кокосами задачу не решить...

Делать Правильное Дело

Конечно, фактор Гиллигана является преувеличением, внесенным для комического эффекта. Но мало кто станет утверждать, что узкое место между идеями и их воплощением в работающих системах полностью исчезло. Даже сегодня стоимость разработки программного обеспечения значительно превосходит стоимость аппаратной части. Обязательно ли программирование должно быть таким сложным?

Рассмотрим ситуацию внимательней. В общем и целом причина сложности разработки программного обеспечения не связана с той ролью, которую оно должно выполнять, – обычно это четко очерченный процесс реального мира. Скорее она связана с отображением задач реального мира на модели, которые могут исполняться компьютерами. А это отображение осуществляется в контексте языков и средств программирования.

Поэтому путь к устранению узкого места программного обеспечения должен лежать, по крайней мере частично, в оптимизации самого программирования путем применения верных инструментов. В этой практической области можно сделать многое – есть ряд чисто искусственных издержек, вызываемых инструментами, которыми мы пользуемся в настоящее время.

Цикл разработки для статических языков

При использовании традиционных статических языков невозможно избежать издержек, связанных с переходом от кода программ к работающим системам: этапы ком-

пиляции и сборки вводят обязательную задержку в процесс разработки. В некоторых условиях разработки обычным делом является ожидание в течение нескольких часов в неделю завершения цикла сборки приложения, написанного на статическом языке. С учетом современной практики разработки, включающей итеративный процесс компиляции, тестирования и повторной компиляции, такие задержки оказываются дорогостоящими и деморализующими (а то и физически мучительными).

Конечно, в разных организациях это может происходить по-своему, а в некоторых областях приложений требования, предъявляемые к производительности, оправдывают задержки в цикле разработки. Но мне приходилось работать в таких условиях разработки на C++, где программисты шутили, что после запуска своих проектов на перекомпиляцию можно идти обедать. Впрочем, это было не совсем шуткой.

Искусственные сложности

Многие традиционные средства программирования таковы, что за деревьями можно легко потерять из виду лес: процедура программирования становится настолько сложна, что плохо видна задача, выполняемая программой в реальном мире. Традиционные языки отвлекают драгоценное внимание на синтаксические проблемы и разработку кода, ведущего учет системных ресурсов. Очевидно, что сложность не является самоцелью – это нужно четко заявить. Тем не менее некоторые используемые в настоящее время инструменты настолько сложны, что сам язык усложняет задачу и затягивает процесс разработки.

Одним языком не угодишь всем

Многие традиционные языки косвенно поощряют однородные, одноязычные системы. Усложняя интеграцию, они препятствуют использованию многоязычных инструментов. Поэтому, не имея возможности выбрать инструмент, более подходящий для конкретной задачи, разработчики часто вынуждены использовать один и тот же язык для всех компонентов приложения. Так как нет языков, которые одинаково хороши во всем, такое ограничение неизбежно наносит ущерб как продукту, так и производительности труда программиста.

Пока наши машины не станут умны настолько, чтобы понимать указания не хуже нас (возможно, не самая разумная из целей), задача программирования останется. Но в данный момент можно достичь существенного прогресса, упростив механику этой задачи. На эту тему я и хочу сейчас поговорить.

И тут появляется Python

Если эта книга достигла целей, которые перед ней ставились, вы должны сейчас хорошо понимать, почему Python называют «управляющим языком нового поколения». В сравнении с аналогичными инструментами, у него есть важные отличия, которые мы наконец можем суммировать:

Tcl

Подобно Tcl, Python может использоваться в качестве встроенного языка расширения. В отличие от Tcl, Python является также полнофункциональным языком программирования. Для многих пользователей средства Python для работы со структурами данных и поддержка программирования в целом открывают возможность применения его в большем числе областей. Tcl продемонстрировал пользу интеграции интерпретируемых языков с модулями C. Python предоставляет ана-

логичные возможности плюс мощный объектно-ориентированный язык: это не просто процессор командных строк.

Perl

Как и Perl, Python можно применять для создания инструментов оболочки, облегчая использование системных сервисов. В отличие от Perl, у Python простой, легко читаемый синтаксис и замечательно ясная архитектура. Благодаря этому некоторым программистам проще пользоваться Python, который лучше подходит для программ, которые должны повторно использоваться или сопровождаться другими программистами. Бесспорно, Perl является мощным инструментом системного администрирования. Но привлекательность Python возрастает с выходом за рамки обработки текста и файлов.

Scheme/Lisp

Подобно Scheme (и Lisp), Python поддерживает динамический контроль типов, инкрементную разработку и метапрограммирование; в нем открыто состояние интерпретатора и поддерживается создание программ на этапе исполнения. В отличие от Lisp, в Python процедурный синтаксис, знакомый пользователям таких основных языков, как C и Pascal. Если нужно, чтобы конечные пользователи писали расширения, это важное преимущество.

Smalltalk

Как и Smalltalk, Python поддерживает объектно-ориентированное программирование (ООП) в контексте весьма динамического языка. В отличие от Smalltalk, Python не включает в систему объектов базовые конструкции управляющей логики программы. Чтобы работать с Python, пользователям не придется осваивать понятие оператора *if* как объекта, получающего сообщения, – Python более традиционен.

Icon

Как и Icon, Python поддерживает множество типов данных и операций высокого уровня, например списки, словари и срезы. В отличие от Icon, Python по существу прост. Программистам (и конечным пользователям) не нужно овладевать эзотерическими понятиями типа перебора с возвратом, чтобы начать работу.

BASIC

Как и у современных структурированных диалектов BASIC, у Python интерпретируемая/интерактивная природа. В отличие от большинства BASIC'ов, в Python есть стандартная поддержка развитых функций программирования, таких как классы, модули, исключительные ситуации, типы данных высокого уровня и общая интеграция с C.

Все эти и другие языки имеют свои достоинства и уникальные сильные стороны – на самом деле Python позаимствовал из таких языков большинство своих характеристик. Цель Python не состоит в том, чтобы заменить все другие языки. Разные задачи требуют разных инструментов, и одной из главных идей Python является разработка в смешанной языковой среде. Но соединившиеся в Python передовые конструкции программирования и средства интеграции делают его выбор естественным в тех областях, о которых рассказывалось в этой книге.

А как насчет того узкого места?

Вернемся к первоначальному вопросу: как можно облегчить работу написания программ? В какой-то мере Python действительно является «просто еще одним компьютерным языком». Несомненно, что с теоретической точки зрения язык Python не

представляет множества радикальных нововведений. Так почему нас должен интересовать Python, когда уже есть столько языков?

Чем интересен Python и что может быть его крупным вкладом в дело разработки программ, так это не особенности синтаксиса или семантики, а взгляд на мир: сочетание инструментов, осуществленное в Python, делает быструю разработку приложений реалистичной целью. Вкратце, Python способствует быстрой разработке благодаря наличию таких характеристик:

- Быстрый цикл разработки
- Объектно-ориентированный язык очень высокого уровня
- Средства интеграции, способствующие многоязыковым разработкам

Более точно, Python борется с узким местом разработки программного обеспечения по четырем направлениям, описываемым в следующих разделах.

Python обеспечивает цикл разработки без промежуточных стадий

Цикл разработки Python значительно короче, чем у традиционных инструментов. В Python нет этапов компиляции и компоновки – программы Python просто импортируют модули на этапе исполнения и пользуются содержащимися в них объектами. Благодаря этому программы Python выполняются сразу после внесения в них изменений. А в тех случаях, когда можно осуществлять динамическую перезагрузку модулей, оказывается возможным изменение и перезагрузка частей выполняющейся программы без ее остановки. Рис. 21.1 показывает воздействие Python на цикл разработки.

Поскольку Python интерпретируется, цикл разработки после изменений в программе происходит быстро. А так как синтаксический анализатор Python в основанных на

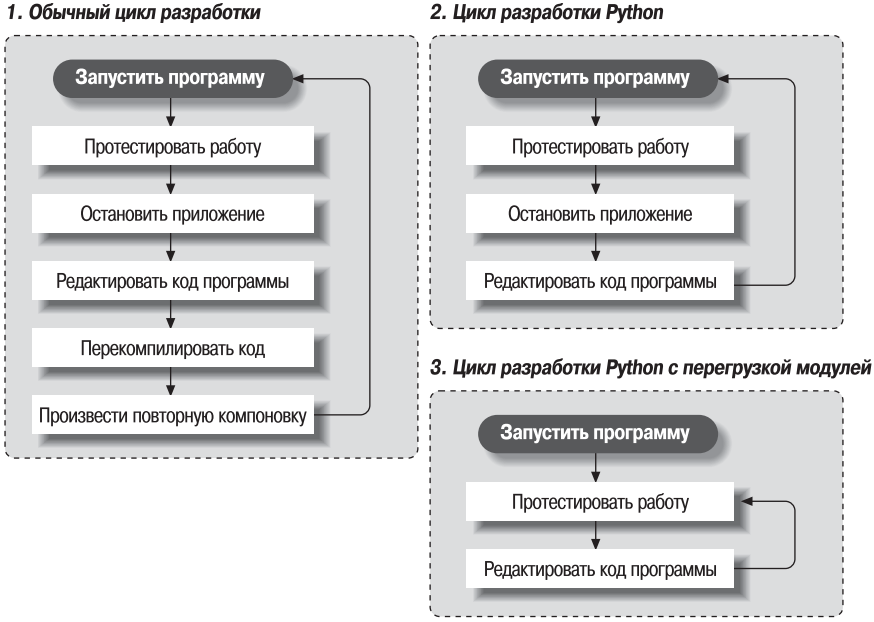


Рис. 21.1. Циклы разработки

Python системах является встроенным, можно легко модифицировать программы во время выполнения. Например, мы уже видели, как разработанные на Python программы GUI позволяют разработчикам изменять код, обрабатывающий нажатие кнопки, при остающемся активным GUI; результат изменения кода можно видеть немедленно, если нажать кнопку снова. Не требуется останавливать программу и компилировать ее заново.

В широком смысле, весь процесс разработки в Python является практикой быстрого прототипирования. Python предоставляет возможность экспериментальной, интерактивной разработки программ и поощряет инкрементную разработку систем путем независимого тестирования компонентов и последующей их сборки. На самом деле мы видели, что можно переключаться между тестированием компонентов (испытаниями программных единиц) и тестированием систем в целом (совместными испытаниями) произвольным образом, как показано на рис. 21.2.

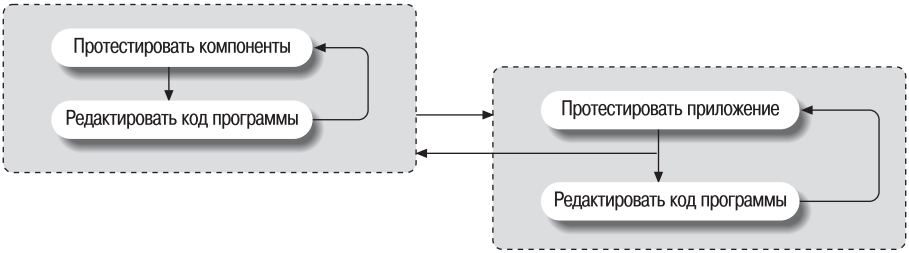


Рис. 21.2. Инкрементная разработка

Python служит «выполняемым псевдокодом»

То, что Python является языком очень высокого уровня, означает, что остается меньше вещей, которые нужно программировать и контролировать. Отсутствия этапов компиляции и сборки в действительности недостаточно для решения собственно проблемы узкого места цикла разработки. Например, наличие интерпретатора C или C++ могло бы обеспечить ускоренный цикл разработки, но все же оказалось бы почти бесполезным для быстрой разработки: сам язык слишком сложен и имеет низкий уровень.

Но так как Python, к тому же, простой язык, кодирование тоже резко ускоряется. Например, динамический контроль типов, встроенные объекты и уборка мусора в значительной мере устраняют необходимость ручного кода для учета ресурсов по сравнению с такими языками низкого уровня, как C и C++. Ввиду видимого отсутствия таких вещей, как объявления типов, управление памятью и реализации стандартных структур данных, программы Python обычно занимают лишь малую долю объема своих эквивалентов на C или C++. Писать и читать приходится меньше, а потому меньше возможности сделать ошибку в программе.

Благодаря отсутствию большей части кода для учета ресурсов программы Python легче понимать, и они лучше отражают действительную задачу, которую призваны решать. А высокий уровень языка Python не только позволяет быстрее реализовывать алгоритмы, но и облегчает изучение языка.

Python – это правильное ООП

Чтобы объектно-ориентированное программирование (ООП) приносило пользу, его применение должно быть простым. Python делает из ООП гибкий инструмент благо-

даря тому, что оно применено в динамическом языке. Важнее, что механизм классов в нем представляет собой упрощенное подмножество C++, и именно такое упрощение обеспечивает полезность ООП в контексте инструмента для быстрой разработки приложений. Например, рассматривая в этой книге классы структур данных, мы видели, что динамический контроль типов в Python позволяет применять один класс ко множеству типов объектов: нам не пришлось писать версии для каждого поддерживаемого типа.

На самом деле в Python так просто применять ООП, что нет причин не использовать его в приложении почти всюду. Модель классов Python обладает достаточно мощными возможностями, позволяющими применять ее в сложных программах, но поскольку доступ к ним прост, это не помеха в решении нашей задачи.

Python способствует созданию гибридных приложений

Как было показано ранее, поддержка Python расширений и встраивания дает возможность использовать его в многоязыковых системах. При отсутствии хороших средств интеграции даже лучший язык быстрой разработки приложений оказывается «закрытым ящиком», не слишком полезным в современных средах разработки. Но средства интеграции Python позволяют применять его в гибридных, многокомпонентных приложениях. Одним из последствий является то, что системы могут одновременно использовать преимущества быстрой разработки на Python и быстрого исполнения традиционных языков типа C.

Хотя использовать Python как самостоятельный инструмент можно, он не навязывает такой режим. Напротив, Python поощряет интегрированный подход к разработке приложений. Поддерживая произвольно соотношение между Python и традиционными языками, Python стимулирует целый ряд парадигм разработки в диапазоне от чистого прототипирования до чистой эффективности. В абстрактном виде это показано на рис. 21.3.

Рис. 21.3. «Ползунок» режима разработки

На левом краю спектра мы оптимизируем скорость разработки. При перемещении к правому краю оптимизируется скорость выполнения. Для каждого проекта оптимальная пропорция лежит где-то посередине. Python позволяет не только выбрать правильную пропорцию для проекта, но и изменить ее позднее произвольным образом, если изменятся потребности:

Идем вправо

Проекты можно начинать на Python с левого конца шкалы и постепенно перемещаться вправо, модуль за модулем, по мере необходимости оптимизировать конечный продукт.

Идем влево

Аналогично можно переносить важные части имеющихся приложений C или C++ в левый конец шкалы, на Python, чтобы обеспечить поддержку программирования и настройки продукта конечным пользователем.

Такая гибкость режима разработки важна в практических условиях. Python оптимизирован по скорости разработки, но одного этого недостаточно. Сами по себе ни C, ни Python не решают проблемы узкого места разработки; объединившись, они могут достичь значительно большего. Как показано на рис. 21.4, помимо самостоятельного использования, одно из наиболее частых применений Python заключается в разбиении систем на *клиентские* (*frontend*) составляющие, получающие преимущества простоты использования Python, и *серверные* (*backend*) модули, требующие производительности статических языков типа C, C++ или FORTRAN.

В любом случае – при добавлении клиентских интерфейсов в существующие системы или разработке их на начальной стадии – такое разделение труда может открыть систему пользователям, не раскрывая ее внутреннего устройства.

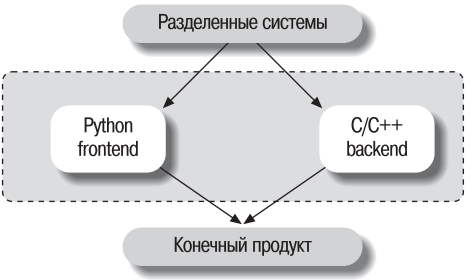


Рис. 21.4. Гибридная архитектура

При разработке новых систем есть также возможность сначала целиком написать ее на Python, а затем при необходимости оптимизировать конечный продукт, перенеся критические для производительности компоненты на компилируемые языки. А так как модули Python и C выглядят для клиентов одинаково, переход на компилированные расширения оказывается прозрачным.

Не во всех ситуациях имеет смысл создавать прототипы. Иногда лучше всего заранее разделить систему на клиентскую часть на Python и сервер на C/C++. И от прототипирования мало пользы при совершенствовании имеющихся систем. Но там, где его можно применять, прототипирование на ранней стадии может оказаться ценным. Путем предварительного создания прототипа на Python можно быстрее показать результаты. Важнее, возможно, что конечные пользователи могут активнее участвовать в ранних стадиях процесса, как показано на рис. 21.5. В результате получаются системы, точнее соответствующие исходным требованиям.

По поводу потопления «Титаника»

Короче, Python в действительности представляет собой больше, чем язык, он предполагает философию разработки. Понятия создания прототипов, быстрой разработки приложений и гибридных приложений, разумеется, не новы. Но в то время как преимущества таких способов разработки широко признавались, существовал недостаток инструментов для применения их на практике без утери мощи программирования. Это один из главных пробелов, которые заполняет архитектура Python: *Python предоставляет простой, но мощный язык для быстрой разработки приложений, наряду со средствами интеграции, необходимыми для применения его в реалистических условиях разработки.*

Такое сочетание, по-видимому, делает Python уникальным среди аналогичных средств. Например, Tcl служит хорошим инструментом интеграции, но не является

Рис. 21.5. Создание прототипов на Python

развитым языком; Perl является мощным языком системного администрирования, но слабым инструментом интеграции. Но тесное соединение в Python мощного динамического языка и интеграции открывает дорогу способам разработки, которые в корне быстрее. Благодаря Python больше не нужно выбирать между быстрой разработкой и быстрым исполнением.

К настоящему времени должно быть ясно, что один язык программирования не может удовлетворять всем задачам разработки. На самом деле потребности иногда оказываются противоречивыми: задачи эффективности и гибкости, по-видимому, всегда будут вступать в конфликт. С учетом высокой стоимости создания программного обеспечения важно выбрать между скоростью разработки и скоростью выполнения. Хотя машинное время дешевле времени программистов, нельзя полностью игнорировать эффективность.

Но имея такой инструмент, как Python, вообще не приходится выбирать между двумя целями. Так же как плотник не станет забивать гвоздь с помощью цепной пилы, так и разработчики программного обеспечения теперь достаточно вооружены, чтобы использовать нужный инструмент для решаемой в данный момент задачи: Python, если важна скорость разработки, компилируемые языки, если доминирует эффективность, и сочетание того и другого, когда цели не определены безусловно.

Более того, не нужно полностью жертвовать повторным использованием кода или полностью переписывать продукт перед поставкой, когда используется быстрая разработка с помощью Python. Быстрая разработка не исключает и второй возможности:

Повторное использование

Так как Python является объектно-ориентированным языком высокого уровня, он содействует написанию повторно используемого программного обеспечения и правильно спроектированных систем.

Готовность продукта к поставке

Так как Python предназначен для использования в системах со смешанными языками, нет необходимости переносить на более эффективные языки сразу весь код.

При типичной разработке на Python можно написать на нем интерфейсную часть и инфраструктуру системы, чтобы облегчить разработку и модификацию, но ядро в целях эффективности все же пишется на C или C++. В таких системах Python называют верхушкой айсберга – та часть, которая видна конечным пользователям, как на рис. 21.6.

Рис. 21.6. «Потопление Титаника» многоязыковыми системами

В такой архитектуре используется лучшее от каждой из частей: ее можно расширить с помощью дополнительного кода Python или модулей расширения C, в зависимости от требований, предъявляемых к производительности. Но это лишь один из многих возможных сценариев разработки с использованием нескольких языков:

Системные интерфейсы

Оформление кода в виде модулей расширения Python облегчает доступ к нему.

Настройка конечным пользователем

Передача программной логики встроенному коду Python позволяет вносить изменения на месте.

Чистое прототипирование

Прототипы Python можно переводить на C все сразу или по частям.

Перенос существующего кода

Перенос существующего кода с C на Python делает его более простым и гибким.

Самостоятельное использование

Конечно, использование Python в самостоятельном виде усиливает существующую библиотеку его инструментов.

Архитектура Python позволяет применять его любым способом, наиболее пригодным для каждого проекта.

Так что же такое Python: продолжение

Как было показано в этой книге, Python является многогранным инструментом, применимым в широком круге областей. Что можно сказать о Python, подводя итоги? К языку Python применимы следующие атрибуты, отражающие лучшие его качества:

- Универсальный
- Объектно-ориентированный
- Интерпретируемый
- Очень высокого уровня

- Открыто разрабатываемый
- Широко переносимый
- Доступный бесплатно
- Живительно ясный

Python полезен как для самостоятельной разработки, так и создания расширений, и оптимизирован для повышения *продуктивности разработчиков* по многим направлениям. Но реальное значение Python должен определить сам читатель. Поскольку Python является универсальным инструментом, то, чем он «является», зависит от того, как вы станете его использовать.

Заключительный анализ...

Я надеюсь, что эта книга что-то рассказала вам о Python, как о языке, так о и выполняемых им ролях. Однако ничто не заменит опыта действительного программирования на Python. Обязательно возьмите какие-нибудь справочные материалы, которые помогут вам на этом пути.

Задача программирования компьютеров, вероятно, всегда будет представлять сложность. Может быть к счастью, но сохранится потребность в умных разработчиках программного обеспечения, искусных в переводе задач реального мира в форму, доступную для выполнения компьютерами, по крайней мере, в обозримом будущем. (В конце концов, будь это просто, нам бы не платили денег. :-)

Но существующие в настоящее время практика разработки и инструменты излишне осложняют наши задачи: многие препятствия, стоящие перед разработчиками являются чисто искусственными. Мы многого достигли в повышении скорости работы компьютеров; пора сосредоточить внимание на скорости разработки. В эпоху постоянно уплотняющихся графиков продуктивность должна иметь первостепенное значение.

Python как инструмент парадигмы совместного использования языков обладает потенциалом стимулировать способы разработки, которые одновременно усиливают выгоды быстрой разработки и традиционных языков. Python не решит всех проблем программной индустрии, но дает надежду на то, что программирование станет проще, быстрее и, по крайней мере, немного приятнее.

Возможно, он не поможет нам совсем покинуть тот остров, но его преимущества над бананами и кокосами несомненны.

Эпилог ко второму изданию

Одним из удовольствий, получаемых при подготовке нового издания такой книги, как эта, является возможность поспорить с самим собой или хотя бы со своими взглядами прошлых лет. Исходя из пятилетней ретроспективы я бы хотел добавить несколько замечаний к первоначальному заключению.

Интеграция – еще не все

В заключении к первому изданию этой книги особо выделялась важность Python как инструмента интеграции. Хотя подчеркнутые там темы по-прежнему сохраняют значимость, я хочу отметить, что не все приложения Python явно исходят из возможности соединения компонентов, написанных на разных языках. Сейчас многие разработчики используют Python в самостоятельном режиме из-за отсутствия у них интегрирующих слоев или не замечая их.

Например, разработчики, пишущие сценарии CGI для Internet с помощью Python, часто кодируют на чистом Python. Где-то дальше в цепочке вызовов происходит обращение к библиотекам C (для доступа к сокетам, базам данных и т. д), но кодировщикам на Python часто не требуется об этом заботиться. На самом деле так оказалось и в моем собственном предшествующем опыте. Во время работы над новыми примерами GUI, операций с системой и Интернетом я долгое время обходился чистым Python. Несколькими месяцами позже я также работал над одной из сред для интеграции Python и C++, но этот интеграционный проект никак не был связан с разработкой примеров для книги на чистом Python. Многие проекты реализуются на чистом Python.

Это не значит, что возможности интеграции Python не являются одним из самых важных его качеств – на самом деле большинство систем Python действительно построено на комбинации Python и C. Однако во многих случаях слой интеграции реализуется однажды горсткой продвинутых разработчиков, а остальные выполняют основной объем программирования, используя только Python. Если вам повезло причислить себя ко второй группе, то общая простота использования Python может оказаться для вас более важной, чем его интегрирующая роль.

Конец войн с Java

В 1995 году сообщество Python воспринимало конфликт между Java и Python как соревнование за умы разработчиков – отсюда и врезка в первом издании с заглавием «Python против Java: первый раунд?». С тех пор это фактически перестало быть проблемой, и я даже полностью убрал эту врезку.

Такое ослабление вражды произошло отчасти потому, что роль Java стала лучше понятна: Java признан как язык *системного программирования*, а не как язык сценариев (scripting language). В сущности, это и предлагалось врезкой. По сложности Java сопоставим с C++ (от которого и произошел), что делает его малопригодным для сценариев, где на первом месте стоит короткий цикл разработки. Это сделано умышленно – Java предназначен для задач, в которых дополнительная сложность может иметь смысл. С учетом большого различия в выполняемых ролях конфликт Python-Java заглох.

Перемирие объявлено также благодаря *JPython* – новой реализации Python. JPython описывался в главе 15 «Более сложные темы Интернета»; вкратце, он интегрирует программы Python и Java таким образом, что возможно создание гибридных приложений: одни их части могут кодироваться на Python, когда оправдано написание сценария, а другие части – на Java, если в них производятся интенсивные вычисления. Это точный аргумент в пользу интеграции с C/C++ из заключения к первому изданию; благодаря JPython те же основания в пользу гибридных систем теперь применяются к приложениям, основанным на Java.

Заявления, делавшиеся в старой врезке по Java, остаются верными – Python проще, более открыт, его легче изучать и применять. Но так и должно быть: в качестве языка сценариев Python естественно дополняет системные языки типа Java и C++, а не конкурирует с ними. Некоторые все же утверждают, что Python лучше подходит для многих приложений, которые сейчас пишутся на Java. Но так же, как и в отношении Python с C и C++, Python и Java лучше всего действуют в одной команде.

Нужно также отметить, что когда я пишу эти слова, Microsoft объявила о новом патентованном языке под названием C#, который, видимо, должен заменить Java в предлагаемых Microsoft системных языках. Кроме того, объявлено о новом переносе Python для среды C#/.NET. Подробности смотрите в главе 15 – примерно этот перенос относится к C#, как JPython относится к Java. Время покажет, начнут ли C# и Java борьбу за умы разработчиков. Но с учетом того, что Python интегрируется с ними

обоими, исход этих битв мегакомпаний представляется несущественным: на этот раз программисты Python могут спокойно наблюдать за ними со стороны.

Нам все еще не удалось выбраться с того острова

Как я упоминал в предисловии к этому изданию, за последние пять лет Python прошел большой путь. Он принят компаниями по всему миру и может теперь гордиться полумиллионной армией пользователей. Несмотря на достигнутый прогресс, остается много работы как в совершенствовании и популяризации Python, так и в упрощении разработки программного обеспечения в целом.

Во время моих передвижений по свету и преподавания Python на курсах и в фирмах я продолжаю встречать много людей, совершенно разочарованных средствами разработки, которыми им приходится пользоваться в своей работе. Некоторые даже меняют работу (или специальность) из-за таких разочарований. Даже спустя большой срок после начала Интернет-революции, разработка программного обеспечения остается более сложным делом, чем это необходимо.

С другой стороны, я также встречаю людей, которые используют Python с таким удовольствием, что не могут себе представить возврата к старому. Они пользуются Python на работе и дома для чистого удовольствия программировать.

Надеюсь, что еще через пять лет я смогу сообщить, что мне попадается значительно больше людей из второй категории, чем из первой. Конечно, Гвидо появился на обложках *Linux Journal* и *Dr. Dobb's* после первого издания этой книги, но нам нужно еще немного потрудиться, чтобы увидеть его на обложке *Rolling Stone*.

Нравоучительная история о Perl и Python

(Следующий текст недавно был послан в телеконференцию *rec.humor.funny* Ларри Хастингсом (Larry Hastings) и воспроизводится здесь с разрешения автора оригинала. Это не значит, что я попустительствую войнам между языками.)

Это прочно засевшая в моем сознании сцена из фильма «Империя наносит ответный удар», перефразированная как ценный моральный урок для честолюбивых программистов.

МЕСТО ДЕЙСТВИЯ: ДАГОБА – ДЕНЬ

Люк, с пристегнутым к его спине Йодой, карабкается вверх по толстой лозе, растущей на болоте, пока не добирается до лаборатории статистики Дагоба. Тяжело дыша, он продолжает свои занятия – grepping, установку новых пакетов, регистрацию в качестве root и замену командных сценариев двухлетней давности, написанных на Python.

ЙОДА: Код! Да. Сила программиста исходит от сопровождаемости его кода. Но берегись Perl. Сжатый синтаксис... больше одного способа сделать это... переменные по умолчанию. Темная это сторона сопровождаемости кода. Легко они текут, быстро приходят, когда ты пишешь код. Однажды пойдя по темному пути, навсегда свяжешь с ним свою судьбу, поглотит тебя он.

ЛЮК: А Perl лучше, чем Python?

ЙОДА: Нет... нет... нет. Быстрее, легче, соблазнительней.

ЛЮК: Но как я узнаю, почему Python лучше, чем Perl?

ЙОДА: Ты узнаешь. Когда пройдет полгода, и код свой ты попробуешь прочесть.

A

Последние изменения в Python

В этом приложении дана сводка важных изменений в версиях Python, выпущенных после выхода первого издания этой книги. Оно разделено на три секции, главным образом потому, что секции по изменениям в версиях 1.6 и 2.0 были составлены на основе замечаний, сопровождающих выпуски версий:

- Изменения, появившиеся в Python 2.0 (и 2.1)
- Изменения, появившиеся в Python 1.6
- Изменения между первым изданием и Python 1.5.2

Python 1.3 был самой свежей версией в момент публикации первого издания книги (октябрь 1996), а Python 1.6 и 2.0 были выпущены как раз перед завершением этого второго издания. Версия 1.6 была последней, выпущенной CNRI, а 2.0 был выпущен BeOpen (двумя работодателями Гвидо до его перехода в Digital Creations); 2.0 добавляет ряд функций в 1.6.

За несколькими заметными исключениями изменения последних пяти лет ввели в Python новые характеристики, но изменения не создали несовместимости. Многие новые характеристики очень полезны, например пакеты модулей, но некоторые, видимо, отвечают прихотям гуру Python, например абстракция списков (list comprehensions), и могут смело игнорироваться всеми остальными. В любом случае, хотя важно следить за эволюцией Python, не следует принимать это приложение слишком серьезно. Откровенно говоря, библиотека приложения и использование инструментов на практике значительно важнее, чем неясные нововведения в язык.

За информацией об изменениях в Python, которые неизбежно произойдут после публикации этого издания, обращайтесь к ресурсам на сайте этой книги (*http://rmi.net/~lutz/about-pp.html*), ресурсам на сайте Python (*http://www.python.org*) или замечаниям к версии, сопровождающим выпуск Python.

Основные изменения в 2.0

В этом разделе перечислены изменения, введенные в Python версии 2.0. Обратите внимание, что расширения сторонних разработчиков для Python 1.5.x или 1.6 нельзя использовать с Python 2.0; эти расширения должны быть перекомпилированы для версии 2.0. Файлы с байт-кодом Python (**.pyc* и **.pyo*) тоже не совместимы для разных версий.

Изменения в базовом языке

В следующих разделах описываются изменения в самом языке Python.

Усовершенствованное присваивание

После почти десятка лет жалоб от программистов С Гвидо сломался и добавил в язык 11 новых операторов присваивания в стиле С:

```
+= -= *= /= %= **= <<= >>= &= ^= |=
```

Оператор `A += B` аналогичен `A = A + B` за исключением того, что `A` вычисляется только один раз (полезно, если это сложное выражение). Если `A` – *изменяемый* (mutable) объект, он может модифицироваться по месту (in place); например, если это список, `A += B` имеет такой же результат, как `A.extend(B)`.

Классы и встроенные типы объектов могут переопределять новые операторы, чтобы реализовать режим изменения по месту; если в объекте не реализован режим изменения по месту, происходит автоматический откат к обычному режиму. Для классов именем метода служит имя соответствующего обычного (non-in-place) оператора, перед которым добавляется «i» (например, `__iadd__` реализует вариант `__add__` для суммирования по месту).

Абстракция списков

Введена новая нотация выражений для списков, элементы которых вычисляются по другому списку (или спискам):

```
[<expression> for <variable> in <sequence>]
```

Например, `[i**2 for i in range(4)]` дает список `[0, 1, 4, 9]`. Это эффективнее, чем применение `map` с lambda, и, по крайней мере в контексте сканирования списков, устраняет некоторые проблемы областей видимости, возникающие с лямбдами (например, использование значений по умолчанию для передачи данных из охватывающей области). Можно также добавить условие:

```
[<expression> for <variable> in <sequence> if <condition>]
```

Например, `[w for w in words if w == w.lower()]` возвращает список слов, не содержащих заглавных букв. Это эффективнее, чем `filter` с лямбдой. Вложенные циклы `for` и более одного `if` также поддерживаются, хотя получаемый при этом код может оказаться таким же сложным, как вложенные построения массивов (map) и лямбды (подробности смотрите в руководствах по Python).

Расширенные операторы import

В операторах импорта теперь допускается использование предложения «as» (например, `import mod as name`), чем избегается присваивание имени импортированного модуля другой переменной. Можно использовать его с операторами `from` и путями пакетов (например, `from mod import var as name`). Слово «as» при этом не стало зарезервированным. (По импорту особых имен файлов, не ложащихся на имена переменных Python, смотрите встроенную функцию `__import__`.)

Расширенные операторы print

Оператор print теперь имеет вариант, при котором вывод направляется в файл, отличный от установленного по умолчанию `sys.stdout`. Например, написать сообщение об ошибке в `sys.stderr` теперь можно так:

```
print >> sys.stderr, "spam"
```

В особом случае, когда выражение, используемое для задания файла, имеет значение `None`, используется текущее значение `sys.stdout` (как без использования `>>`). Конечно, всегда можно писать в файловые объекты, например `sys.stderr`, с помощью их метода

write; данное расширение просто добавляет форматирование, осуществляемое оператором print (например, преобразование строк, пробелы между элементами).

Возможность сбора циклического мусора

Python теперь снабжен сборщиком мусора, который может прослеживать циклические ссылки между объектами Python. Он не заменяет подсчет ссылок (и в действительности зависит от правильного подсчета ссылок), но может решить, что группа объектов составляет цикл, если все их счетчики ссылок учтены в ссылках друг на друга. Новый модуль с именем gc позволяет управлять параметрами уборки мусора; параметр в сценарии Python «configure» позволяет включать или отключать сборку мусора. (Смотрите замечания к версии 2.0 или библиотечное руководство, чтобы проверить, включена ли эта функция по умолчанию; так как периодическое выполнение этого дополнительного этапа сборки мусора снижает производительность, решение о том, включать ли его по умолчанию, не принято.)

Отдельные изменения в библиотеке

Ниже частично перечислены изменения в стандартной библиотеке, введенные в версии Python 2.0. Полное описание изменений смотрите в примечаниях к версии 2.0.

Новая функция zip

Введена новая функция zip: zip(seq1,seq2,...) эквивалентно map(None,seq1,seq2,...), когда последовательности имеют одинаковую длину. Например, zip([1, 2, 3], [10, 20, 30]) возвращает [(1,10), (2,20), (3,30)]. Если списки имеют разную длину, длина результата определяется самым коротким списком.

Поддержка XML

Новый стандартный модуль pyexpat предоставляет интерфейс к парсеру XML Expat. Новый стандартный пакет модулей xml предоставляет разнообразный код поддержки XML в трех (пока) подпакетах: xml.dom, xml.sax и xml.parsers.

Новый модуль веб-броузера

В новом модуле webbrowser сделана попытка предоставить независимый от платформы API для запуска веб-броузера. (Смотрите также сценарий LaunchBrowser в конце главы 4 «Более крупные системные примеры, часть 1».)

Изменения в API интеграции Python/C

Была обеспечена переносимость на 64-разрядные платформы под Linux и Win64, в особенности для нового процессора Intel Itanium. Была также добавлена поддержка больших файлов для Linux64 и Win64.

Изменения в сборке мусора привели к созданию двух новых позиций (slots) в объектах, tp_traverse и tp_clear. Введение сокращенного присваивания привело к созданию новой позиции для каждого оператора изменения по месту. GC API создает новые требования для контейнерных типов, реализованные в модулях расширения C. Смотрите *Include/objimpl.h* в дистрибутиве исходного кода Python.

Изменения для Windows

В модуль os добавлены новые вызовы popen2, popen3 и popen4.

Вызов os.popen теперь гораздо шире может использоваться в Windows 95 и 98. Для организации этого вызова в Windows 9x Python внутренне использует программу w9xpopen.exe, находящуюся в корневом каталоге установки Python (это не самостоятельная программа). Смотрите детали в статье Microsoft Knowledge Base Q150956.

Права администратора больше не требуются при установке Python под Windows NT или Windows 2000. Под Windows инсталляция теперь по умолчанию выполняется в каталог *\Python20* на томе по умолчанию (например, *C:\Python20*), а не в *\Program Files\Python-2.0*, как прежде.

Программа установки под Windows больше не запускает отдельного инсталлятора Tcl/Tk; теперь необходимые файлы Tcl/Tk устанавливаются прямо в каталог Python. Если Tcl/Tk уже установлен, теряется некоторое дисковое пространство (примерно 4 Мбайт), но устраняются конфликты между разными установками Tcl/ Tk, а Python может с меньшим трудом обеспечить для Tcl/Tk нахождение всех своих файлов.

Главные изменения в версии 1.6

В этом разделе перечислены изменения, введенные в Python версии 1.6; большинство из них вошло также в версию 2.0.

Несовместимость

Метод append для списков больше нельзя вызывать с несколькими аргументами. Раньше он добавлял один кортеж, составленный из всех аргументов, но эта возможность не была документирована. Для добавления кортежа нужно написать: `l.append((a, b, c))`.

Методы сокетов connect, connect_ex и bind требуют ровно один аргумент. Раньше можно было написать: `s.connect(host, port)`, хотя так не было задумано; сейчас нужно писать: `s.connect((host, port))`.

Функции str и repr теперь различаются еще чаще. Для длинных целых чисел str больше не добавляет «L»; `str(1L)` дает «1», а раньше было «1L», а `repr(1L)` по-прежнему возвращает «1L». Для чисел с плавающей точкой repr теперь дает точность 17 цифр, что обеспечивает сохранение точности (на используемой в настоящее время аппаратуре).

Некоторые библиотечные функции и инструменты переведены в категорию устаревших, в том числе такие широко использовавшиеся средства, как find. Модуль string теперь просто представляет интерфейс к новым методам строк, но с учетом того, что он используется почти всеми модулями Python, написанными к этому времени, весьма маловероятно, что его не станет.

Изменения в базовом языке

В следующих разделах описываются изменения в самом языке Python.

Строки Unicode

Python стал поддерживать строки Unicode (то есть 16-битные символы). В версии 1.6 введен новый базовый тип данных (строка Unicode), новая встроенная функция unicode и несколько C API для работы со строками Unicode и кодировками. Константы строк Unicode имеют префикс «u», подобно неформатированным строкам (например, `u"..."`). Подробности смотрите в файле *Misc/unicode.txt* дистрибутива Python или зайдите на сайт *http://starship.python.net/crew/lemburg/ unicode-proposal.txt*.

Характеристики версии Python 2.1 Alpha[1]

Если немного подождать, то набор характеристик Python изменится – как погода в Колорадо. Перед самой сдачей этого издания в печать было объявлено о выходе альфа-версии Python 2.1. В числе прочего в ней есть такие нововведения:

- К функциям можно присоединять произвольные атрибуты: именам атрибутов функций просто осуществляется присваивание, чтобы связать с функцией дополнительную информацию (то, что кодировщики делали с помощью форматированных строк документации).
- Новое богатое расширение сравнения теперь позволяет классам перегружать отдельные операторы сравнения специальными методами (например, `__lt__` перегружает проверки `<`) вместо выполнения всех проверок в одном методе `__cmp__`.
- Структура вывода предупреждений предоставляет интерфейс к сообщениям, выдаваемым при использовании устаревших функций (например, модуля `regex`).
- Система сборки Python переделана для использования пакета `Distutils`.
- Новый атрибут `sys.displayhook` дает пользователям возможность настраивать способ вывода объектов в интерактивном приглашении.
- Построчный ввод/вывод файлов (метод файла `readline`) стал работать значительно быстрее, а новый метод файла `xreadlines` читает по одной строке в циклах `for`.
- Кроме того: изменена модель приведения чисел в расширениях C, модули могут теперь устанавливать имя `__all__`, указывающее имена, экспортируемые ими предложением `from *`, модуль `ftplib` теперь по умолчанию устанавливает «passive» для лучшей работы с брандмауэрами и т. д.
- Другие усовершенствования, такие как статически вложенные области и слабые ссылки, в момент выхода альфа-версии все еще прорабатывались.

Конечно, как всегда, следует обратиться к веб-странице этой книги (*http://www.rmi.net/~lutz/about-pp.html*) и замечаниям к выпуску версий Python 2.1 и следующих, чтобы узнать о дальнейшем развитии Python, которое несомненно произойдет сразу после сдачи этой вставки в издательство.

[1] В настоящее время Python уже преодолел эту стадию развития и в ходу стабильная версия 2.2. – *Примеч. науч. ред.*

Методы строк

Многие функции модуля `string` теперь доступны как методы объектов строк. Например, можно теперь сказать `str.lower()` вместо импорта модуля `string` и выражения `string.lower(str)`. Эквивалентом для `string.join(sequence,delimiter)` служит `delimiter.join(sequence)`. (То есть `" ".join(sequence)` используется для имитации `string.join(sequence)`.)

Новая машина регулярных выражений (внутренняя)

Новая машина регулярных выражений SRE полностью обратносовместима с прежней и вызывается с помощью того же интерфейса (модуля). Таким образом, для кодирования поиска по-прежнему используется интерфейс модуля `re`, который не изменился, но реализован он с использованием SRE. Можно явно вызвать прежнюю машину, им-

портировав pre, либо машину SRE, импортировав sre. SRE быстрее pre и поддерживает Unicode (что было главной причиной разработки еще одной базовой машины регулярных выражений).

Синтаксис вызова функций в стиле apply

Вместо функции apply можно использовать специальный синтаксис вызова функций: f(*args, **kwds) эквивалентно apply(f, args, kwds). Можно также использовать такие разновидности, как f(a1, a2, *args, **kwds), или опустить одно или другое (например, f(*args), f(**kwds)).

Основание при преобразовании строк в числа

Встроенные функции int и long принимают необязательный второй аргумент, задающий основание преобразования, но только если первый аргумент является строкой. Это сделало string.atoi и string.atol устаревшими. (string.atof уже была таковой.)

Лучшее диагностирование ошибок локальных имен

Когда локальная переменная известна компилятору, но оказывается неопределенной при использовании, возбуждается новая исключительная ситуация UnboundLocalError. Это класс, производный от NameError, поэтому код, перехватывающий NameError, должен по-прежнему работать. Цель в том, чтобы точнее диагностировать ситуации, как в таком примере:

```
x = 1
def f():
    print x
    x = x+1
```

Ранее возбуждалась непонятная ошибка NameError в операторе print.

Перегрузка оператора принадлежности

Появилась возможность перегрузки оператора in путем определения метода __contains__. Обратите внимание на использование аргументов в обратном порядке: x in a выполняет a.__contains__(x) (поэтому имя не __in__).

Отдельные изменения в библиотечных модулях

В этом разделе перечислены некоторые изменения в стандартной библиотеке Python.

distutils

Новый; инструменты для распространения модулей Python.

zipfile

Новый; чтение и запись архивов zip (модуль gzip работает с файлами gzip).

unicodedata

Новый; доступ к базе данных Unicode 3.0 database.

_winreg

Новый; доступ к реестру Windows (модуль без _ находится в разработке).

socket, httplib, urllib

Расширены введением поддержки защищенных сокетов OpenSSL (только для Unix).

_tkinter

Поддержка Tk версий с 8.0 по 8.3.

`string`

Этот модуль больше не использует встроенный модуль C `strop`, а пользуется новыми методами строк, предоставляя прозрачную поддержку как обычных строк, так и строк Unicode.

Некоторые изменения в утилитах

В этом разделе перечислены некоторые изменения в инструментарии Python.

IDLE

Полностью переработан. Дополнительная информация – на домашней странице IDLE на at *http://www.python.org*.

Tools/i18n/pygettext.py

Эквивалент Python для средства извлечения текста сообщений `xgettext`, используемого при интернационализации приложений, написанных на Python.

Главные изменения между версиями 1.3 и 1.5.2

В этом разделе описаны важные изменения в языке, библиотеке, инструментах и C API Python между первым изданием книги (Python 1.3) и версией Python 1.5.2.

Изменения в базовом языке

В следующих разделах описаны изменения в самом языке Python.

Псевдоприватные атрибуты классов

Python теперь обеспечивает протокол корректировки имен, который скрывает имена атрибутов, используемые классами. Внутри оператора класса имя вида `_X` автоматически изменяется Python на `_Class__X`, где `Class` является именем класса, определяемого оператором. Поскольку впереди добавляется имя содержащего класса, эта функция сужает возможность конфликта имен при расширении или встраивании существующих классов. Обратите внимание, что это вовсе не механизм «приватных» переменных, а просто функция локализации имен классов, ограничивающая конфликты имен в иерархиях и совместно используемом пространстве имен экземпляра объекта внизу цепочки наследования атрибутов.

Классы исключительных ситуаций

Исключительные ситуации теперь могут иметь вид объектов классов (и экземпляров классов). Это сделано с целью поддержки категорий исключительных ситуаций. Так как предложение `except` теперь будет соответствовать возбужденной исключительной ситуации, если в нем назван соответствующий класс или любой из его суперклассов, то задание суперклассов позволяет операторам `try` перехватывать целые категории без явного перечисления всех их членов (например, перехват исключительной ситуации суперкласса числовой ошибки будет также перехватывать конкретные классы числовых ошибок). Стандартные встроенные исключительные ситуации Python являются теперь классами (а не строками) и образуют неглубокую иерархию классов; смотрите подробности в руководстве по библиотеке.

Импорт пакетов

Операторы импорта могут теперь ссылаться на пути к каталогам на компьютере, используя синтаксис с точками для обозначения пути. Например:

```
import directory1.directory2.module                    # и использовать путь
from   directory1.directory2.module import name        # и использовать "name"
```

Оба оператора загружают модуль, вложенный на два уровня в пакетах (каталогах). Самое левое имя пакета в пути импорта (directory1) должно быть каталогом, перечисленным в пути поиска модулей Python (sys.path, инициализируемый PYTHONPATH). Соответственно путь оператора import обозначает подкаталоги, в которые нужно пройти. Пути предотвращают конфликты имен модулей при установке на одной машине нескольких систем Python, которые предполагают найти собственную версию модуля с тем же именем (в противном случае побеждает первая, найденная в PYTHONPATH).

В отличие от старого модуля ni, который заменен этой функцией, поддержка нового пакета всегда доступна (без выполнения специального импорта) и требует, чтобы каждый каталог пакета вдоль пути импорта содержал файл модуля *__init__.py* (возможно, пустой), идентифицирующий каталог как пакет и служащий в качестве его пространства имен при прямом импортировании. С пакетами лучше использовать from, чем import, так как для вызова импортированных объектов после import нужно повторять полный путь.

Новый оператор assert

Python 1.5 ввел новый оператор:

```
assert test [, value]
```

означающий то же, что и:

```
if __debug__:
    if not test:
        raise AssertionError, value
```

Этот оператор предназначен в основном для отладки, но может использоваться для задания ограничений программы (например, проверки ввода при входе в функции).

Изменения в списке зарезервированных слов

В список зарезервированных слов Python добавлено слово «assert»; слово «access» изъято (теперь оно действительно устарело).

Новые методы словарей

Несколько вспомогательных методов добавлено во встроенный объект словаря, чтобы избавиться от ручных циклов: D.clear(), D.copy(), D.update() и D.get(). Первые два метода очищают и копируют словари соответственно. D1.update(D2) эквивалентен циклу:

```
for k in D2.keys(): D1[k] = D2[k]
```

D.get(k) возвращает D[k], если тот существует, или None (либо дополнительный второй аргумент), если ключ не существует.

Новые методы списков

В объектах списков появился новый метод pop, загружающий и удаляющий из списка последний элемент:

```
x = s.pop()     ...то же, что два оператора...     x = s[-1]; del s[-1]
```

и extend для присоединения списка элементов к концу по месту:

```
s.extend(x)     ...то же, что...                   s[len(s):len(s)] = x
```

В метод pop можно также передать индекс удаляемого элемента (равный по умолчанию −1). В отличие от append, в extend передается целый список, и каждый из его элементов добавляется в конец.

Константы неформатируемых («raw») строк

Для поддержки регулярных выражений в Windows Python позволяет записывать строковые константы в виде r"...\..." , которые действуют как обычные строки за исключением того, что Python не трогает обратные слэши в строке. Они остаются как литеральные символы \ и не интерпретируются Python как особые escape-коды.

Тип комплексных чисел

Python теперь поддерживает константы комплексных чисел (например, 1+3j) и операции комплексной арифметики (обычные математические операторы плюс модуль cmath со многими функциями модуля math для комплексных чисел).

Вывод циклических объектов не приводит к дампу памяти

Объекты, создаваемые кодом типа L.append(L), теперь обнаруживаются интерпретатором и выводятся особым образом. В прошлом при попытке вывода циклических объектов интерпретатор входил в рекурсивный цикл, что в конечном счете вело к дампу памяти.

raise без аргументов: повторное возбуждение ошибки

Оператор raise без исключительной ситуации или аргументов с дополнительными данными заставляет Python заново возбудить последнюю сгенерированную и не перехваченную исключительную ситуацию.

Форматы raise для классов исключительных ситуаций

Так как исключительные ситуации могут теперь быть строковыми объектами или классами и экземплярами классов, можно использовать любой из следующих форматов оператора raise:

```
raise string           # соответствует except с таким же объектом строки
raise string, data     # то же с дополнительными данными

raise class, instance  # соответствует except с классом или его суперклассом
raise instance         # то же, что: raise instance.__class__, instance

raise                  # заново возбудить последнюю исключительную ситуацию
```

Можно также использовать следующие три формата, сохраняемые для обратной совместимости с прежними версиями, где встроенные исключительные ситуации были строками:

```
raise class            # то же, что: raise class() (и: raise class, instance)
raise class, arg       # то же, что: raise class(arg)
raise class, (arg,...) # то же, что: raise class(args...)
```

Оператор возведения в степень X ** Y

Новый бинарный оператор ** вычисляет левый операнд в степени правого операнда. Он действует сходным со встроенной функцией pow образом.

Обобщенное присваивание последовательностей

В присваиваниях (операторах = и других контекстах присваивания) можно теперь присваивать последовательность в правой части списку или кортежу слева (например, (A,B) = seq, [A,B] = seq). Ранее типы последовательностей должны были совпадать.

Большая скорость

Python 1.5 показал быстродействие почти вдвое более высокое, чем его предшественники, по тесту *Lib/test/pystone.py*. (В других тестах я видел почти трехкратное превосходство в скорости.)

Изменения в библиотеках

В следующих разделах описываются изменения, произведенные в стандартной библиотеке Python.

dir(X) теперь применим к большему числу объектов

Встроенная функция `dir` теперь сообщает атрибуты модулей, классов и экземпляров классов, а также встроенных объектов, таких как списки, словари и файлы. Не требуется использовать такие члены, как `__methods__` (но по-прежнему можно).

Новые преобразования: int(X), float(X), list(S)

Встроенные функции `int` и `float` теперь принимают строковые аргументы и преобразуют строки в числа точно так же, как `string.atoi/atof`. Новая встроенная функция `list(S)` преобразует любую последовательность в список, весьма сходным с более старым и неясным приемом `map(None, S)` образом.

Новый модуль re регулярных выражений

Новый модуль регулярных выражений `re` предлагает полноценный поиск регулярных выражений в стиле Perl. Подробности смотрите в главе 18 «Текст и язык». Более старый модуль `regex`, описанный в первом издании, по-прежнему доступен, но считается устаревшим.

splitfields/joinfields превратились в split/join

Функции `split` и `join` из модуля `string` были обобщены и могут выполнять те же действия, что и первоначальные `splitfields` и `joinfields`.

Постоянное хранение: unpickler больше не вызывает __init__

Начиная с Python 1.5 обратное из последовательного вида преобразование (загрузчик) модуля `pickle` более не вызывает метод класса `__init__` для воссоздания сериализованных объектов экземпляров классов. Это значит, что классам более не требуется устанавливать значения по умолчанию для всех аргументов конструкторов, используемых с сохраняемыми объектами. Чтобы заставить Python вызвать метод `__init__` (как он делал это ранее), классы должны обеспечить специальный метод `__getinitargs__`; подробности в руководстве по библиотеке.

Объект pickler написан на C: cPickle

Реализация модуля `pickle` на C теперь стандартно входит в Python. Она называется `cPickle` и, по сообщениям, во много раз быстрее первоначальной. Если есть модуль `shelve`, он автоматически загружает `cPickle` вместо `pickle`.

anydbm.open теперь должен получить «c» как второй аргумент, чтобы работать в прежнем режиме

Чтобы открыть файл DBM в режиме «создать новый или открыть существующий для чтения и записи», вторым аргументом в `anydbm.open` передается «c». Это изменилось начиная с Python 1.5.2; при передаче «c» теперь происходит то, что раньше было при

отсутствии второго аргумента (второй аргумент теперь по умолчанию равен «r» – только для чтения). Это не влияет на `shelve.open`.

Модуль rand заменен модулем random

Модуль `rand` объявлен устаревшим, вместо него надлежит использовать `random`.

Различные изменения в Tkinter

Tkinter стал переносимым на все основные платформы (Windows, X, Macs), обеспечив на них свойственный системе внешний вид. В GUI Tkinter произошел ряд изменений:

Объекты StringVar вызывать нельзя

Метод `__call__` для объектов класса `StringVar` был удален в Python 1.4; это значит, что необходимо явно вызывать их методы `get()/set()`, а не обращаться к ним с аргументами или без.

ScrolledText изменен

Графический элемент `ScrolledText` подвергся небольшим изменениям интерфейса в Python 1.4, которые были отменены в версии 1.5, по-видимому, из-за возникшей несовместимости.

Сеточный менеджер геометрии

Tkinter теперь поддерживает новый менеджер геометрии Tk `grid`. Для его использования нужно вызывать метод `grid` объектов графических элементов (подобно `pack`, но передаются номера рядов и колонок, а не условия).

Новый сайт документации Tkinter

Фредрик Лунд (Fredrik Lundh) теперь сопровождает прекрасный комплект документации по Tkinter на *http://www.pythonware.com*, где есть справочники и учебники.

Изменение интерфейса модуля CGI

Изменился интерфейс CGI. Прежний интерфейс `FormContent` объявлен устаревшим, и его заменил интерфейс объекта `FieldStorage`. Детали смотрите в руководстве по библиотеке.

site.py, user.py и PYTHONHOME

Эти сценарии автоматически выполняются при запуске Python, используются для настройки первоначальных путей. Подробности в руководствах по библиотеке.

Присваивание os.environ[key] вызывает putenv

Присваивание по ключу в словаре `os.environ` теперь изменяет соответствующую переменную окружения в окружении C. Оно запускает вызов функции библиотеки C `putenv`, чтобы отразить изменения в слоях интегрированного кода C, а также в окружении всех дочерних процессов, порожденных программой Python. `putenv` теперь также открыта в модуле `os` (`os.putenv`).

Новый кортеж sys.exc_info()

Новая функция `exc_info()` в модуле `sys` возвращает кортеж значений, соответствующий `sys.exc_type` и `sys.exc_value`. Эти более старые имена обращаются к одной глобальной исключительной ситуации; `exc_info` специфична для вызвавшего процесса.

Новый модуль operator

Есть новый стандартный модуль `operator`, который предоставляет функции, реализующие большинство встроенных операторов выражений Python. Например, `operator.add(X,Y)` делает то же, что `X+Y`, но так как модуль операторов экспортирует функции, иногда удобно использовать их в таких вещах, как `map`, чтобы не создавать функций и не использовать лямбда-форму.

Изменение в инструментарии

В следующих разделах описываются главные изменения, относящиеся к инструментам Python.

JPython (или Jython): компилятор Python в Java

Новая система JPython является альтернативной реализацией Python, компилирующей программы Python в байт-код виртуальной машины Java (JVM), и предоставляет ловушки для интеграции Python и программ Java. Смотрите главу 15 «Более сложные темы Интернета».

Переносы под MS-Windows: COM, Tkinter

Интерфейсы COM в переносах Python под Windows были существенно развиты с момента их описания в первом издании (тогда это называлось «OLE»); смотрите главу 15. Python теперь также поставляется для Windows как самоинсталлирующаяся программа со встроенной поддержкой интерфейса Tkinter, файлов в стиле DBM и др. Сегодня для установки достаточно двойного щелчка.

Развитие SWIG, теневые классы C++

Система SWIG стала главным инструментом разработчиков расширений с новыми «теневыми классами», служащими оболочками классов C++. Смотрите главу 19 «Расширение Python».

Zope (бывший Bobo): объекты Python для Сети

Эта система публикации объектов Python в веб стала инструментом, популярным среди программистов CGI и веб-сценаристов в целом. Смотрите раздел, посвященный Zope, в главе 15.

HTMLgen: создание HTML из классов Python

Это средство генерации правильных файлов HTML (описаний структур веб-страниц) из деревьев объектов классов Python достигло своей зрелости. Смотрите главу 15.

PMW: графические элементы Python высокого уровня для Tkinter

Система PMW предоставляет мощные графические элементы высокого уровня для основанных на Tkinter GUI в Python. Смотрите главу 6 «Графические интерфейсы пользователя».

IDLE: GUI интегрированной среды разработки

Python теперь поставляется с интерфейсом «point-and-click» («укажи-и-щелкни») под названием IDLE. Написанный на Python с использованием библиотеки GUI Tkinter, IDLE либо находится в каталоге Tools исходного кода библиотеки, либо устанавливается автоматически с самим Python (в Windows смотрите пункт IDLE в меню Python, открываемом кнопкой Пуск). IDLE предлагает текстовый редактор с расцвет-

кой синтаксиса, графический отладчик, броузер объектов и др. Если у вас в Python включена поддержка Tk и вы привыкли к развитым интерфейсам разработчика, IDLE предоставит богатую функциями альтернативу традиционной командной строке Python. На сегодняшний день IDLE не предоставляет средства построения GUI.

Развитие других инструментов: PIL, NumPy, Database API

Системы обработки графики PIL и численного программирования NumPy заметно развились. Выпущен также API баз данных для Python. Смотрите главу 6 и главу 16 «Базы данных и постоянное хранение».

Изменения в API интеграции Python/C

В следующих разделах описаны изменения, произведенные в Python C API.

Единый файл заголовков Python.h

Все используемые Python символы экспортируются теперь в одном файле заголовков *Python.h*; в большинстве случаев не требуется импортировать другие файлы заголовков.

Единый файл библиотеки C libpython*.a

Весь код интерпретатора Python теперь при сборке Python оформлен в одном библиотечном файле. Например, для Python 1.5 встраивание требует теперь компоновки только с *libpython1.5.a* (вместо прежней схемы с четырьмя библиотеками и файлами *.o*).

«Великое (величайшее?) переименование» завершено

Все открытые символы Python теперь имеют префикс «Py».

Поддержка потоков, множественные интерпретаторы

API нескольких новых инструментов обеспечивают лучшую поддержку потоков при встраивании Python. Например, есть средства для завершения Python (`Py_Finalize`) и для создания «множественных интерпретаторов» (`Py_NewInterpreter`).

Обратите внимание, что порождение потоков языка Python может быть жизненноважной альтернативой потокам уровня C, а нескольких пространств имен часто оказывается достаточно для изоляции имен, используемых в независимых системных компонентах; обеими схемами легче управлять, чем множественными интерпретаторами и потоками. Но в некоторых многопоточных программах полезно также иметь по одному экземпляру системных модулей и структур для каждого потока и здесь удобно использовать множественные интерпретаторы (например, если в каждом потоке нет своего экземпляра, импорт может обнаружить уже загруженный модуль в таблице `sys.modules`, если его импортировал другой поток). Подробности смотрите в руководствах по новым C API.

Новая документация по Python C API

Вместе с Python поставляется новое справочное руководство, описывающее основные средства и режимы C API. Оно еще не вполне завершено, но служит хорошей отправной точкой.

B

Прагматика

Это приложение служит очень кратким введением в некоторые подробности применения Python на уровне инсталляции и содержит список ресурсов Интернета по Python. Дополнительные сведения по темам, неполно здесь освещенным, можно найти в других источниках:

- За дополнительными сведениями по установке обратитесь к различным текстовым файлам README в дистрибутиве примеров на прилагаемом к этой книге CD, а также к файлам README и другой документации, сопровождающей дистрибутивы Python и другие пакеты, которые есть на CD. В частности, файлы README в каталогах *Examples* и *Examples\PP2E* могут содержать документацию к дереву примеров и детали установки Python, не повторяющиеся в данном приложении.
- Другую подробную информацию по выполнению программ Python в целом можно найти в руководствах по Python, имеющихся на CD этой книги, или в книге O'Reilly вводного уровня «*Learning Python*».
- За дополнительной информацией по собственно базовому языку Python обратитесь к стандартным руководствам Python, имеющимся на прилагаемом CD, и книгам O'Reilly «*Learning Python*» и «*Python Pocket Reference*».
- За дополнительной информацией любого рода, относящейся к Python, обратитесь на *http://www.python.org*. На этом сайте можно взять электронную документацию Python, загрузить интерпретатор, также здесь есть поисковый механизм и ссылки чуть ли не на все сайты, касающиеся Python. За ссылками на информацию о данной книге вернитесь к предисловию.

Установка Python

В этом разделе дается обзор относящихся к инсталляции подробностей – инструкции по установке интерпретатора Python на вашем компьютере.

Windows

Детали установки Python зависят от платформы и описываются в только что перечисленных ресурсах. Но в общих чертах, пользователи Windows обнаружат исполняемый модуль самоинсталляции Python на прилагаемом к книге CD (см. каталоги верхнего уровня Python 1.5.2 и 2.0). Нужно просто сделать двойной щелчок по программе инсталляции и отвечать «yes», «next» или «default», чтобы пройти стандартную установку для Windows. Если попутно будет запрос, нужно ли устанавливать Tcl/Tk, следует ответить утвердительно.

После установки в меню Программы кнопки Пуск появится группа Python. В нее входят пункты для запуска интегрированного интерфейса разработки IDLE и сеанс консоли командной строки, просмотр стандартных руководств Python и др. Руководства Python в формате HTML устанавливаются вместе с интерпретатором и открываются локально в веб-броузере при выборе соответствующего пункта меню.

Python также регистрирует себя для открытия файлов Python в Windows, поэтому для запуска сценариев Python можно дважды щелкнуть по ним в файловом менеджере Windows. Запускать сценарии Python можно также вводом командной строки `python file.py` в ответ на приглашение командной строки DOS при условии, что каталог, в котором находится программа интерпретатора Python `python.exe`, добавлен в переменную PATH оболочки DOS (см. ниже разделы по конфигурации и выполнению).

Обратите внимание, что в стандартный пакет Python для Windows входит полная поддержка Tkinter. Для выполнения GUI Tkinter под Windows не нужно устанавливать другие пакеты или выполнять какие-либо другие действия по инсталляции, достаточно установить Python. Все необходимые компоненты Tkinter устанавливаются программой самоинсталляции Python, и Python автоматически находит нужные компоненты без дополнительной настройки окружения. В установку под Windows также входит расширение `bsddb` для поддержки файлов в стиле DBM.

Если вы планируете заниматься специфической для Windows работой, например разработкой COM, то может понадобиться установить дополнительный пакет расширений `win32all` (имеющийся на прилагаемом CD, а также на *http://www.python.org*). Этот пакет регистрирует Python для Active Scripting, предоставляет классы-оболочки MFC и интеграцию с COM и др. (см. главу 15 «Более сложные темы Интернета»). Учтите также, что дистрибутивы Python, которые можно получить в других местах (например, дистрибутив ActivePython от ActiveState, *http://www.activestate.com*), могут содержать как Python, так и пакет расширений Windows.

Unix и Linux

Python может оказаться уже установленным на этих платформах (сейчас он часто входит в стандартную установку Linux); посмотрите в своих каталогах */usr/bin* и */usr/local/bin*, нет ли там интерпретатора Python. Если нет, то Python обычно устанавливается на этих платформах из пакета `rpm` (автоматически устанавливающего выполняемые модули и библиотеки Python) или из пакета дистрибутива исходного кода (который распаковывается и компилируется локально на компьютере). Компиляция Python из исходного кода под Linux тривиальна – обычно нужно ввести две-три простых командных строки. Детали есть в файлах README вверху дистрибутива исходного кода Python и документации Linux по `rpm`.

Macintosh и другие

Пожалуйста, посмотрите детали установки и использования в документации, относящейся к переносам под Macintosh. Для других платформ, вероятно, потребуется найти перенос на *http://www.python.org* и обратиться к соответствующим примечаниям по установке переноса или документации.

Дистрибутив примеров книги

В этом разделе кратко обсуждается дистрибутив исходного кода примеров в книге и освещаются детали использования примеров.

Пакет с примерами книги

Каталог компакт-диска *Examples\PP2E* является пакетом модулей Python, содержащим файлы исходного кода для всех примеров, представленных в книге (и другие). Пакет *PP2E*, в свою очередь, содержит вложенные пакеты модулей, которые разделяют файлы примеров на подкаталоги по темам. Можно запускать файлы прямо с CD или копировать каталог *PP2E* на жесткий диск своей машины (копирование позволит редактировать файлы и даст возможность Python сохранять их скомпилированный байт-код для более быстрого запуска).

В любом случае каталог, содержащий корень *PP2E*, обычно должен быть указан в пути поиска модулей Python (обычно в переменной окружения PYTHONPATH). Это единственный элемент, который нужно добавить в путь Python; операторы импорта в примерах книги всегда представляют собой пути импорта пакетов относительно корневого каталога *PP2E* за исключением случаев, когда импортируемый модуль находится в одном каталоге с импортирующим.

Кроме того, в пакете примеров находятся сценарии для преобразования символов перевода строки в файлах примеров в формат Unix и из него (на CD они имеют формат DOS), установки атрибутов файлов, разрешающих запись (полезно после перетаскивания файлов в Windows) и другие. Дополнительные сведения об использовании дерева пакетов и утилит смотрите в файлах README вверху деревьев каталогов *Examples* и *PP2E*.

Выполнение сценариев для запуска демонстрационных программ

На верхнем уровне пакета *Examples\PP2E* есть самонастраивающиеся сценарии Python, позволяющие запускать основные примеры книги, даже если вы не настраивали окружение. То есть они должны работать, даже если не устанавливать переменные оболочки PATH или PYTHONPATH. Эти два сценария, PyDemos и PyGadgets, представлены в главе 8 «Обзор Tkinter, часть 2» и более полно описаны в предисловии и файлах README на CD. В большинстве случаев эти сценарии можно запустить прямо с прилагаемого к книге CD с помощью двойного щелчка в GUI менеджера файлов (в предположении, что Python установлен).

Настройка окружения

Этот раздел знакомит с деталями настройки окружения для Python и описывает установки, которые воздействуют на программы Python.

Переменные оболочки

Следующие переменные оболочки (среди других) обычно важны при работе с Python:

PYTHONPATH

Путь поиска модулей Python. Если она установлена, то используется для нахождения модулей на этапе выполнения, когда их импортирует программа Python – Python ищет файл импортируемого модуля или пакета в каждом каталоге, перечисленном в PYTHONPATH слева направо. Python обычно также автоматически осуществляет поиск в исходном каталоге сценария и в каталоге исходного кода стандартной библиотеки Python, поэтому их добавлять не нужно. Совет: посмотрите `sys.path` в интерактивном режиме, чтобы узнать, как действительно установлен путь.

PYTHONSTARTUP

Необязательный файл инициализации Python. Если используется эта переменная, установите ее равной полному имени файла с кодом Python (модуля), который должен выполняться при каждом запуске интерактивного интерпретатора командной строки Python. Это удобный способ импорта модулей, которыми вы часто пользуетесь при интерактивной работе.

PATH

Переменная, содержащая путь поиска операционной системой исполняемых файлов. Используется для нахождения файлов программ при вводе их имен в командной строке без указания полного пути к каталогу. Поместите в нее каталог, содержащий исполняемый файл интерпретатора `python` (иначе каждый раз придется вводить путь к нему).

Кроме того, пользователям, работающим на платформах, отличных от Windows, может потребоваться установить переменные для использования Tkinter, если не удается найти место установки Tcl/Tk. Установите переменные `TK_LIBRARY` и `TCL_LIBRARY`, чтобы они указывали на локальные каталоги файлов библиотек *Tk* и *Tcl*.

Установки конфигурации

В каталоге *Examples\PP2E\Config* на прилагаемом CD содержатся примеры файлов конфигурации для настройки переменных Python с комментариями. В Windows NT можно установить эти переменные в GUI системных настроек (подробнее об этом ниже); в Windows 98 можно устанавливать их в пакетных файлах DOS, которые, в свою очередь, можно запускать из *C:\autoexec.bat*, чтобы гарантировать их установку при каждом запуске компьютера. Например, в моем файле *autoexec* есть строка:

```
C:\PP2ndEd\examples\PP2E\Config\setup-pp.bat
```

которая вызывает файл, содержащий строки для добавления Python в системную переменную PATH и корня пакета примеров книги в PYTHONPATH:

```
REM PATH %PATH%;c:\Python20
PATH %PATH%;c:\"program files"\python

set PP2EHOME=C:\PP2ndEd\examples
set PYTHONPATH=%PP2EHOME%;%PYTHONPATH%
```

Выберите (то есть снимите `REM`) одну из первых двух строк в зависимости от того, где установлен у вас Python, – первая строка предполагает установку 2.0 по умолчанию, а вторая предполагает установку Python 1.5.2. Измените также значение `PP2EHOME`, чтобы оно указывало на каталог, содержащий на вашей машине корень примеров *PP2E* (тот, который показан, действует на моей машине). Под Linux мой файл начального запуска *~/.cshrc* вызывает файл *setup-pp.csh*, который выглядит аналогично:

```
setenv PATH $PATH:/usr/bin
setenv PP2EHOME /home/mark/PP2ndEd/examples
setenv PYTHONPATH $PP2EHOME:$PYTHONPATH
```

Однако синтаксис установки переменных зависит от оболочки (подробности в каталоге CD *PP2E\Config*). Назначение переменной оболочки PYTHONPATH списка каталогов, как здесь, действует на большинстве платформ и служит типичным способом настройки пути поиска модулей. На некоторых платформах есть другие способы установки пути поиска. Вот несколько советов по конкретным платформам:

Windows

Порт для Windows позволяет использовать реестр Windows в дополнение к установке PYTHONPATH в DOS. В некоторых версиях Windows вместо изменения *C:\autoexec.bat* и перезагрузки можно установить путь, выбрав Control Panel, затем значок System, щелкнуть по вкладке Environment Settings и ввести в появившемся диалоговом окне PYTHONPATH и тот путь, который нужен (например, *C:\mydir*). Такая установка становится постоянной, как при помещении ее в *autoexec.bat*.

JPython

Для JPython, реализации Python на Java, путь может иметь вид аргументов командной строки – `Dpath` в команде Java, используемой для запуска программы, или присваиваний `python.path` в файлах регистрации Java.

Конфигурирование из программы

В любом случае `sys.path` представляет путь поиска в сценариях Python и инициализируется из настроек путей в среде и стандартных значений по умолчанию. Это обычный список строк Python, который программы Python могут изменять, чтобы динамически настраивать путь поиска. Чтобы расширить путь поиска изнутри Python, выполните:

```
import sys
sys.path.append('mydirpath')
```

Так как переменные окружения доступны программам Python через встроенный словарь `os.environ`, сценарий Python может также выполнить что-нибудь вроде `sys.path.append(os.environ['MYDIR'])` и добавить в результате каталог, содержащийся в переменной окружения MYDIR, к пути поиска модулей Python на этапе исполнения. Поскольку `os.pathsep` задает символ, используемый вашей платформой в качестве разделителя в путях каталогов, а `string.split` умеет разбивать строки по разделителям, то группа операторов:

```
import sys, os, string
path = os.environ['MYPYTHONPATH']
dirs = string.split(path, os.pathsep)
sys.path = sys.path + dirs
```

добавляет к пути поиска модулей все имена из списка, указанного в MYPYTHONPATH, точно так же, как Python обычно делает для PYTHONPATH. С помощью таких изменений `sys.path` можно динамически настраивать путь поиска модулей из сценария. Однако они действуют, только пока выполняется создавшая их программа или сеанс Python, поэтому в большинстве случаев лучше устанавливать PYTHONPATH.

Выполнение программ Python

Код Python можно вводить в интерактивном приглашении `>>>`, выполнять из программы C или размещать в текстовых файлах и выполнять их. Есть несколько способов выполнения кода в файлах:

Запуск из командной строки

Файл Python всегда можно запустить, введя команду вида `python file.py` в системной оболочке или окне консоли, если программа интерпретатора Python находится в системном пути поиска. В Windows можно ввести эту команду в окне консоли MS-DOS; в Linux воспользуйтесь xterm.

Выполнение щелчком

В зависимости от платформы обычно можно запускать файлы программ Python двойным щелчком по значку в пользовательском интерфейсе менеджера файлов. В Windows, например, файлы Python *.py* автоматически регистрируются таким образом, что можно запускать их при щелчке (как и файлы *.pyc* и *.pyw*).

Выполнение путем импорта и перезагрузки

Файлы можно также выполнять путем их импорта как интерактивно, так и из другого файла модуля. Чтобы повторно выполнить код файла модуля, не выходя из Python, выполните вызов типа `reload(module)`.

Выполнение файлов в IDLE

Для многих программистов адекватной средой разработки в Python служит открытое окно консоли и одно или несколько окон текстового редактора. Для других GUI среды разработки Python предоставляет IDLE – (Integrated Development Environment, в действительности назван в честь Eric Idle из «Monty Python»). В ней можно также выполнять существующие файлы программ или разрабатывать новые системы с чистого листа. IDLE написан на Python/Tkinter и потому переносим на Windows, X Windows (Unix) и Macintosh. Он бесплатно поставляется в качестве стандартного инструмента вместе с интерпретатором Python. В Windows IDLE автоматически устанавливается вместе с Python; смотрите подраздел «Windows» в разделе «Установка Python» выше в этом приложении.

IDLE позволяет редактировать, отлаживать и выполнять программы Python. Он выделяет цветом синтаксис редактируемого кода Python, имеет броузер объектов, позволяющий проходить через системные объекты параллельно с кодом программы и предоставляет интерфейс отладчика Python типа «point-and-click». Подробности смотрите в тексте подсказки IDLE и на странице на *http://www.python.org*. Можно также просто попробовать поработать с ним, так как большинство его интерфейсов интуитивны и легко понятны. Единственное, чего в IDLE, по-видимому, сегодня не хватает, это возможности создания GUI с помощью мыши (но благодаря простоте Tkinter такие средства менее важны в работе с Python).

Выполнение файлов в Pythonwin

Pythonwin является еще одной общедоступной IDE open source для Python, но предназначена только для платформ Windows. В ней используется интеграция с MFC, ставшая доступной программистам Python в ориентированном на Windows пакете расширений Python `win32all`, который описан в главе 15. На самом деле Pythonwin представляет собой нечто вроде иллюстрации применения этих инструментов для Windows. Pythonwin поддерживает редактирование и запуск исходного кода, подобно IDLE (между этими системами произошло некое перекрестное опыление). Эта среда не обладает всеми функциями или переносимостью IDLE, но предлагает разработчикам для Windows собственные средства. Загрузите и установите пакет расширений `win32all` для Windows и поэкспериментируйте с Pythonwin. Этот пакет также можно найти на CD, прилагаемом к книге.

Выполнение Python из других IDE

Если вы привыкли к более сложным средам разработки, посмотрите продукты Visual Python от Active State (*http://www.activestate.com*) и продукты PythonWorks от PythonWare (*http://www.pythonware.com*). Оба они находятся в развитии, когда я это пишу, и обещают предоставить программистам Python развитые интегрированные наборы инструментов. Например, в планы ActiveState входит поддержка разработки программ Python как в Microsoft Visual Studio, так и в среде приложений Mozilla, а также перенос Python в среду C#/.NET. Продукты

PythonWare поддерживают графическую разработку интерфейсов и набор инструментов разработчика.

На других платформах существуют дополнительные способы запуска программ Python (например, сбрасывание файлов на значки в Mac). Вот еще несколько советов пользователям Unix и Windows:

Пользователям Unix и Linux

Файлы модулей Python можно сделать непосредственно исполняемыми, добавив специальную строку типа `#!/usr/bin/python` в начало файла и предоставив право выполнения файла командой *chmod*. В результате становится возможным запуск файлов Python путем ввода их имен (например, *file.py*), как если бы они были скомпилированными исполняемыми модулями. Подробности смотрите в главе 2 «Системные инструменты».

Пользователям Windows

Если при запуске программы Python в менеджере файлов выскакивает окошко, это может быть окно консоли, открываемое Python для представления стандартных входных/выходных потоков программы. Если вы хотите оставить это окно открытым, чтобы посмотреть на вывод, добавьте в конец программы вызов `raw_input()`; в результате будет сделана остановка до нажатия клавиши Enter. Если вы пишете собственный GUI и вообще не хотите, чтобы всплывала консоль, дайте файлу исходного кода расширение *.pyw* вместо *.py*.

Windows NT и 2000

Файл сценария Python можно запустить из приглашения командной строки, просто введя имя его файла (например, *file.py*). На этих платформах сценарий правильно запускается с интерпретатором Python без специальной первой строки `#!`, необходимой для непосредственного запуска файлов в Unix. Для выполнения командных строк Python на платформах Windows 9x необходимо добавлять слово «python» перед именем файла и обеспечивать нахождение пути к *python.exe* в установке PATH (как описано выше). На всех платформах Windows можно также запускать программы Python щелчком по имени файла в менеджере файлов Windows.

Интернет-ресурсы по Python

Наконец, в табл. B.1 перечислены некоторые наиболее полезные сайты, где находятся информация и ресурсы Python. Почти на все из них можно попасть с домашней страницы Python (*http://www.python.org*), и большинство адресов может со временем поменяться, поэтому обращайтесь за самыми свежими подробностями к домашней странице Python.

Таблица B.1. Ссылки Python в Интернете

Ресурс	Адрес
Главный сайт Python	*http://www.python.org*
FTP-сайт Python	*ftp://ftp.python.org/pub/python*
Телеконференция Python	*comp.lang.python (python-list@cwi.nl)*
Главный сайт O'Reilly	*http://www.oreilly.com*
O'Reilly Python DevCenter	*http://www.oreillynet.com/python*
Сайт книги	*http://www.rmi.net/~lutz/about-pp2e.html*
Сайт автора	*http://www.rmi.net/~lutz*

Таблица B.1 (продолжение)

Ресурс	Адрес
Почтовый список поддержки Python	*mailto:python-help@python.org*
Электронные руководства Python	*http://www.python.org/doc*
FAQ Python	*http://www.python.org/doc/FAQ.html*
Группы специальных интересов Python	*http://www.python.org/sigs*
Поиск ресурсов Python	*http://www.python.org/search*
Starship (библиотека)	*http://starship.python.net*
Vaults of Parnassus (библиотека)	*http://www.vex.net/parnassus*
Jpython	*http://www.jpython.org*
SWIG	*http://www.swig.org*
Tk	*http://www.scriptics.com*
Zope	*http://www.zope.org*
ActiveState (инструменты)	*http://www.activestate.com*
PythonWare (инструменты)	*http://www.pythonware.com*

C

Python и C++

В этом приложении кратко суммированы некоторые различия между классами Python и C++. Систему классов Python можно рассматривать как подмножество системы классов C++. Сравнение с Modula 3 могло быть более тесным, но C++ сегодня является доминирующим языком ООП. В Python модель классов намеренно упрощена – классы являются просто *объектами* с присоединенными к ним *атрибутами*, которые могут быть связаны с другими объектами классов. Поддерживается создание множественных экземпляров, настройка (customization) путем наследования атрибутов и перегрузка операторов, но объектная модель Python сравнительно облегченная. Вот несколько конкретных различий между Python и C++:

Атрибуты

Реального различия между данными и методами в Python нет; те и другие просто определяют именованные *атрибуты* экземпляров или классов, привязанные к функциям или другим видам объектов. Атрибуты являются именами, прикрепляемыми к объектам, доступ к которым происходит путем квалификации: `object.attribute`. Методы являются просто атрибутами класса, присваиваемыми функциям, обычно создаваемым вложенными операторами `def`; члены являются просто именами атрибутов, присваиваемыми объектам других видов.

Создание объектов классов

Операторы классов создают объекты классов и присваивают их именам. Операторы, выполняющие присваивание именам внутри оператора `class`, создают атрибуты класса; классы наследуют атрибуты всех других классов, перечисленных в заглавной строке их оператора `class` (поддерживается множественное наследование; оно обсуждается ниже).

Создание объектов экземпляров

Вызов объекта класса, как если бы он являлся функцией, создает новый объект *экземпляра* класса. Экземпляр возникает с пустым пространством имен, наследующим имена из пространства имен класса; присваивание атрибутам экземпляра (например, атрибутам `self` в функциях методов классов) создает атрибуты в экземпляре.

Удаление объекта

Классы и экземпляры (и все содержащиеся в них данные) автоматически удаляются, когда на них больше нет ссылок. Нет оператора `new` (вместо него вызываются классы), а оператор Python `del` лишь убирает одну ссылку, в отличие от *delete* в C++.

Создание членов

Атрибуты классов и экземпляров, подобно простым переменным, начинают существовать, когда осуществляется присваивание им, не объявляются заранее и

могут ссылаться на объект любого типа (они могут даже в разное время указывать на объекты разных типов).

Наследование

Наследование в Python обычно срабатывает при поиске значения по имени атрибута: получив выражение вида `object.attribute`, Python ищет в дереве объектов пространства имен, начиная с `object` и выше, первое появление имени `attribute`. Поиск согласно наследованию происходит также, когда к объектам применяются операторы выражений и операции типов. Новый независимый поиск согласно наследованию осуществляется для каждого вычисляемого выражения `object.attribute` – даже выражения `self.attr` внутри метода функции снова ведут поиск `attr` в объекте экземпляра, на который указывает `self`, и выше.

Информация времени выполнения

Классы Python на этапе исполнения являются *объектами*, находящимися в памяти, их можно передавать в программе, обеспечивая своего рода источник информации о типах времени выполнения (например, одна функция может генерировать экземпляры произвольных классов, переданных в качестве аргумента). Объекты классов и экземпляров несут информацию интерпретатора (например, словарь атрибутов `__dict__`), а функция Python `type` позволяет проверять тип объекта. Атрибуты `__class__` объектов экземпляров ссылаются на класс, которым они созданы, а атрибуты `__bases__` объектов классов дают суперклассы класса (базовые классы).

«this» указатель

Эквивалент Python для указателя C++ на экземпляр `this` является первым аргументом, добавляемым в вызовы функций методов (обычно принято называть его `self`). В вызове он обычно задается неявно, но явно используется в методах: нет скрытой области видимости экземпляра для неквалифицированных имен. Методы Python суть просто функции, вложенные в оператор `class`, которые получают в качестве самого левого параметра предполагаемый объект экземпляра.

Виртуальные методы

В Python все методы и члены-данные являются виртуальными (`virtual`) в том смысле, как это понимается в C++: отсутствует понятие разрешения атрибутов на этапе компиляции исходя из типа объекта. Все квалификации атрибутов (`object.name`) разрешаются на этапе исполнения исходя из типа квалифицируемого объекта.

Чисто виртуальные методы

Методы, вызываемые суперклассом, но не определенные в нем, соответствуют понятию C++ о «чисто виртуальных» методах: методах, которые должны переопределяться в подклассах. Так как Python не компилируется статически, для объявления такого случая не нужно специального синтаксиса, как в C++. Вызовы неопределенных методов возбуждают на этапе исполнения исключительную ситуацию ошибки имени, которую можно перехватывать в операторах `try`.

Статические члены

Не существует объявления `static` данных класса; вместо этого присваивания, вложенные в оператор `class`, генерируют имена атрибутов, связываемые с этим классом и совместно используемые всеми его экземплярами.

Закрытые (private) члены

Нет понятия о действительном ограничении доступа к атрибутам; все данные и методы суть *public* в смысле C++. Скрытие атрибутов регулируется соглашениями, а

не синтаксисом: ограничения C++ `public`, `private` и `protected` не применимы (но смотрите новую функцию вида `_X` локализации имени класса в приложении А «Последние изменения в Python»).

Интерфейсы const

Объекты могут быть немодифицируемыми, а имена – нет: эквивалент модификатора C++ `const` отсутствует. Ничто не мешает изменить имя или объект в методе, а методы могут изменять модифицируемые аргументы (например, объект `self`). Традиции и здравый смысл заменяют лишний синтаксис.

Параметры-ссылки

Нет прямого аналога параметрам-ссылкам C++. Методы Python могут возвращать несколько значений в виде кортежа и могут изменять передаваемые объекты, если они модифицируемы (например, путем присваивания атрибутам объекта или изменения списков или словарей по месту). Но нет совмещения имен в вызове и заголовке функции: аргументы передаются путем присваивания, благодаря чему создаются совместно используемые ссылки на объекты.

Перегрузка операторов

Операторы перегружаются методами со специальными именами: синтаксиса, похожего на `operator+`, нет, но эффект аналогичен. Например, атрибут класса `__add__` перегружает (перехватывает и выполняет) применение оператора + к экземплярам класса; `__getattr__` примерно похож на перегрузку `->` в C++. Произвольные выражения требуют кодирования правосторонних методов (например, `__radd__`).

Шаблоны

В Python осуществляется динамический контроль типов – имена служат ссылками на произвольные объекты, и понятие объявления типа отсутствует. Шаблоны C++ неприменимы, и в них нет необходимости. Классы и функции Python в целом могут применяться к любому типу объектов, реализующему интерфейсные протоколы (операции и операторы), предполагаемые кодом класса. Их объект не обязательно должен иметь определенный тип данных.

Дружественность

В Python все является дружественным. Нет понятия закрытости, поэтому любой класс имеет доступ к внутренним элементам другого.

Перегрузка функций

В Python полиморфизм основан на вызовах виртуальных методов: тип квалифицирующего объекта определяет, что делают его методы. Так как в Python типы аргументов не объявляются (контролируются динамически), нет ничего похожего на перегрузку функций в C++, когда происходит обращение к различным версиям функции в зависимости от типов данных ее аргументов. Можно явным образом проверять типы и длину списка аргументов, а не писать отдельные функции для каждой комбинации типов (см. встроенную функцию `type` и формат аргументов функции `*args`).

Множественное наследование

Множественное наследование обозначается в коде перечислением более одного суперкласса в круглых скобках за строкой заголовка оператора class. При использовании множественного наследования Python просто берет первое вхождение атрибута при поиске в глубину слева направо в дереве суперклассов. Конфликты множественного наследования Python разрешает именно таким способом, а не рассматривает их как ошибки.

Виртуальное наследование

Понятие C++ виртуальных базовых классов не вполне применимо в Python. Экземпляр класса Python является одним словарем пространства имен (с указателем на класс для доступа к унаследованным атрибутам). Классы помещают атрибуты в словарь экземпляра класса путем присваивания. Благодаря такой структуре каждый атрибут существует лишь в одном месте – словаре экземпляра. Для унаследованных атрибутов поиск в дереве суперклассов однозначно разрешает ссылки.

Конструкторы

Python выполняет только один метод `__init__`, найденный в дереве наследования объектов. Он не выполняет автоматически конструкторы всех доступных классов; при необходимости конструкторы других классов вызываются вручную. Но это не труднее, чем задавать аргументы конструкторов суперклассов в C++. Деструкторы Python (`__del__`) выполняются при сборке мусора экземпляров (то есть удалении), а не в ответ на вызов *delete*.

Операторы области видимости

Операторы C++ области видимости типа `Superclass::Method` используются для расширения наследуемых методов и устранения неоднозначности в наследовании. В Python ближайшим эквивалентом служит `Superclass.Method`, квалификация объекта класса. Это не требуется для разрешения конфликтов наследования, но может использоваться для отмены правила поиска по умолчанию и обращения к суперклассам при расширении методов.

Указатели методов

В Python ссылки на методы являются *объектами* и не используют специального синтаксиса; их можно передавать, хранить в структурах данных и т. д. Объекты методов бывают двух видов: *связанные* методы (когда экземпляр известен) являются парами экземпляр–метод и вызываются впоследствии как простые функции; *несвязанные* методы служат просто ссылками на объект метода функции и при вызове требуют явного задания экземпляра.

Конечно, в Python есть дополнительные характеристики классов, отсутствующие в C++, например *протоколы метаклассов*: `__setattr__` можно применять для создания альтернативных интерфейсов, а указатель класса `__class__` можно переустановить, чтобы динамически поменять тип класса. Кроме того, атрибуты класса можно произвольно менять на этапе исполнения: классы являются просто объектами с присоединенными именами атрибутов.

Python отличается от C++ во многих отношениях помимо моделей классов. Например, в Python нет ни объявлений типов, ни этапов компиляции и компоновки; в Python нельзя перегрузить `=`, как это можно сделать в C++ (присваивание не является в Python оператором); указатели, одно из центральных понятий в программировании на C и C++, совершенно отсутствуют в Python (хотя ссылки на объекты частично обладают их свойствами). Вместо указателей в программах Python используются объекты первого класса (first-class objects), которые автоматически размещаются и удаляются.

Большинство этих различий происходит из-за того, что Python проектировался для ускорения разработки, а не ускорения выполнения; значительная часть дополнительного синтаксиса C++ помешала бы целям Python. Полное введение в классы и базовый язык Python имеется в книге O'Reilly «Learning Python».

Алфавитный указатель

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

А

Б

В

Г

Д

З

И

К

M

П

P

C

T

X

Ш