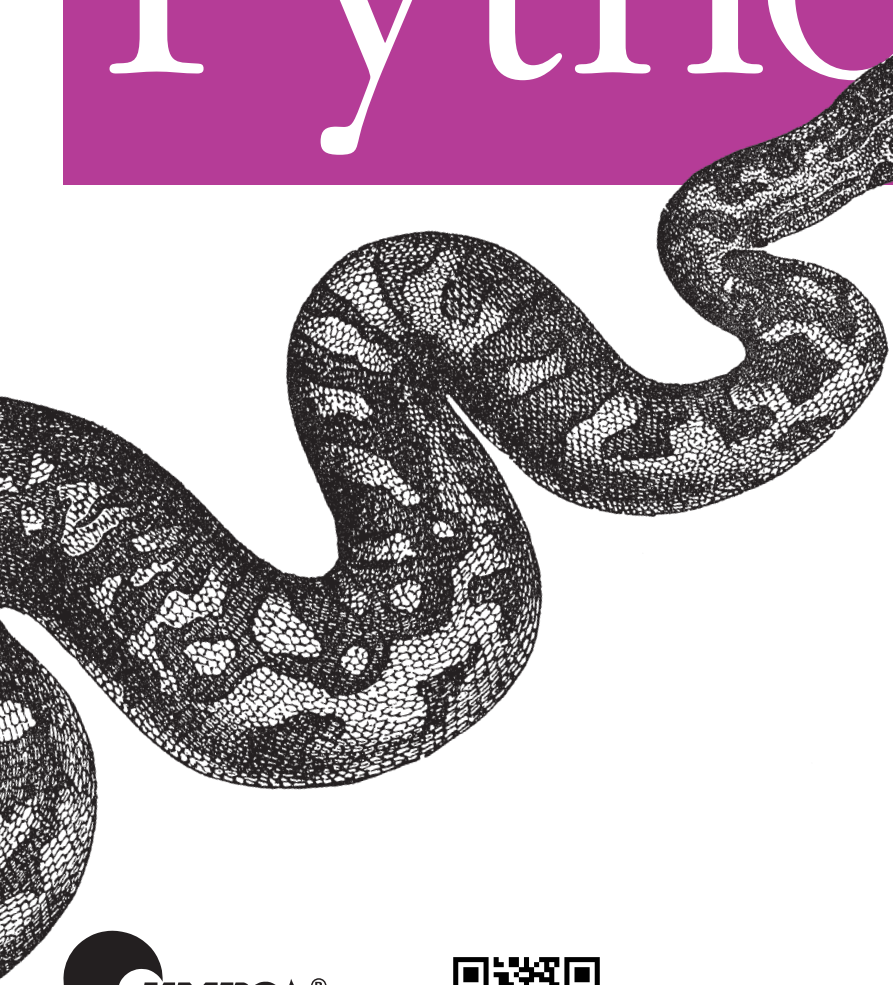
Эффективное объектно-ориентированное программирование

4-е издание
охватывает Python 3.X

Программирование на

Python

том I



Символ
O'REILLY

Марк Лутц

Programming Python

Forth Edition

Mark Lutz

O'REILLY®

Программирование на Python

том I

Четвертое издание

Марк Лутц

Санкт-Петербург — Москва
2011

Марк Лутц

Программирование на Python, том I, 4-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Лутц М.

Программирование на Python, том I, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 992 с., ил.

ISBN 978-5-93286-210-0

Вы овладели основами Python. Что дальше? Эта книга представляет собой подробное руководство по применению этого языка программирования в основных прикладных областях – системном администрировании, создании графических интерфейсов и веб-приложений. Исследуются приемы работы с базами данных, программирования сетевых взаимодействий, создания интерфейсов для сценариев, обработки текста и многие другие.

Издание описывает синтаксис языка и методики разработки, содержит большое количество примеров, иллюстрирующих типичные идиомы программирования и корректное их применение. Кроме того, исследуется эффективность Python в качестве инструмента разработки программного обеспечения, в отличие от просто инструмента «создания сценариев».

В четвертое издание включено описание новых особенностей языка, библиотек и практических приемов программирования для Python 3.X. Примеры, представленные в книге, опробованы под третьей альфа-версией Python 3.2.

Можно смело утверждать, что это обстоятельная и всесторонняя книга предназначена быть первой ступенью на пути овладения мастерством разработки приложений на языке Python.

ISBN 978-5-93286-210-0
ISBN 978-0-596-15810-1 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 31.07.2011. Формат 70×100 1/16.
Печать офсетная. Объем 62 печ. л.

Оглавление

Предисловие

«А теперь нечто совершенно иное...»

В этой книге исследуются способы применения языка программирования Python в типичных прикладных областях и в реально возникающих задачах. Эта книга рассказывает, что можно *делать* с языком Python после того, как вы овладели его основами.

Эта книга предполагает, что читатель еще только начинает знакомиться с рассматриваемыми в книге прикладными областями – графическими интерфейсами, Интернетом, базами данных, системным программированием и так далее, – и представляет каждую из них, начиная с самых азов, выполняя роль *учебника*. Попутно книга ставит своей целью познакомить читателя с часто используемыми инструментами и библиотеками, но не с основами языка. Таким образом, данная книга является ресурсом, позволяющим читателю получить более глубокое понимание роли языка Python в практике программирования.

Дополнительно в этой книге исследуется пригодность языка Python на роль *инструмента разработки программного обеспечения*, в отличие от просто инструмента «создания сценариев». Многие примеры, представленные в этой книге, подобраны в соответствии с этой целью – среди них вы найдете примеры постепенной разработки клиентов электронной почты, опирающиеся на тысячи строк программного кода. Создание подобных крупномасштабных приложений всегда будет непростым делом, но мы покажем, насколько быстрее и проще создаются такие приложения, когда они разрабатываются на языке Python.

Это четвертое издание было дополнено представлением новых особенностей языка, библиотек и практических приемов программирования для Python 3.X. В частности, примеры, представленные в книге, выполняются под управлением интерпретатора версии Python 3.1 – наиболее свежей версии Python на момент написания этих строк. Непосредственно перед публикацией книги все основные примеры были опробованы под третьей альфа-версией Python 3.2, но вообще говоря, они должны сохранить свою работоспособность при использовании любой версии Python из линейки 3.X. Кроме того, это издание было реорганизовано с целью упорядочить прежний материал и добавить описание новых инструментов и тем.

Поскольку среди читателей этой книги будут и те, кому в руки первым попало это издание, и те, кто знаком с предыдущими изданиями, я хочу в этом предисловии более подробно остановиться на целях и задачах этой книги, прежде чем перейти к программному коду.

Об этой книге

Данная книга является учебником по применению языка Python для решения наиболее типичных задач в различных прикладных областях. В ней рассказывается о применении языка Python в системном администрировании, для создания графических интерфейсов и веб-приложений и исследуются приемы программирования сетевых взаимодействий, взаимодействий с базами данных, обработки текста, создания интерфейсов для сценариев и во многих других областях. Несмотря на то, что на протяжении всей книги используется язык Python, тем не менее основное внимание будет уделяться не основам языка, а приемам решения практических задач.

Экосистема этой книги

Диапазон тем, обсуждаемых в этой книге, позволяет ее рассматривать, как второй том двухтомника, который должен быть дополнен третьим томом. Важно помнить, что эта книга описывает особенности разработки приложений и является продолжением книги «Изучаем Python»[1], рассматривающей основы языка, знание которых совершенно необходимо для чтения этой книги. Поясним, как связаны эти книги между собой:

- «Изучаем Python» – подробно описывает основы программирования на языке Python. Основное внимание уделяется базовым особенностям языка Python, знание которых является необходимой предпосылкой для чтения этой книги.
- «Программирование на Python» – эта книга охватывает практические приемы программирования на языке Python. Основное внимание в этой книге уделяется библиотекам и инструментам, и предполагается, что читатель уже знаком с основами языка.
- «Python Pocket Reference» – краткий справочник, в котором охватываются некоторые подробности, отсутствующие в данной книге. Этот справочник не может использоваться в качестве учебника, но он позволит вам быстро отыскивать описание тех или иных особенностей.

В некотором смысле, эта книга является аналогом «Изучаем Python», но в ней раскрываются не основы языка, а основы прикладного программирования. Это последовательный учебник, в котором не делается

[1] Марк Лутц «Изучаем Python», 4 издание, СПб.: Символ-Плюс, 2010.

никаких предположений об уровне вашей подготовки, и каждая тема начинает рассматриваться с самых азов. Например, при изучении темы разработки веб-приложений вы приобретете все необходимые знания, которые позволят вам создавать простые веб-сайты, и получите представление о более развитых фреймворках и инструментах, которые пригодятся вам с ростом ваших потребностей. Обсуждение особенностей конструирования графических интерфейсов также идет по нарастающей – от простого к сложному.

Дополнением к этой книге может служить книга «Python Pocket Reference» (карманный справочник по языку Python), где предоставляется дополнительная информация о некоторых особенностях, не рассматриваемых в данной книге, и которая может служить отличным справочником. Книга «Python Pocket Reference» является всего лишь справочником, и в ней практически отсутствуют примеры и пояснения, но она может служить отличным дополнением к книгам «Изучаем Python» и «Программирование на Python». Поскольку текущее четвертое издание «Python Pocket Reference» содержит информацию об обеих основных версиях Python, 2.X и 3.X, оно также может использоваться читателями, выполняющими переход между этими двумя версиями Python (подробнее об этом чуть ниже)[1].

Чего нет в этой книге

Из-за распределения областей рассмотрения по книгам, о которых упоминалось выше, область применения этой книги имеет следующие ограничения:

- Она не раскрывает основы языка Python
- Она не предназначалась для использования в качестве справочника по особенностям языка

Первый из этих двух пунктов отражает тот факт, что освещение основ языка является исключительной прерогативой книги «Изучаем Python», и если вы совершенно не знакомы с языком Python, я рекомендую сначала прочитать книгу «Изучаем Python», прежде чем приниматься за эту книгу, так как здесь предполагается знание основ языка.

---

[1] Я являюсь автором всех трех книг, упомянутых в этом разделе, и это дает мне возможность контролировать их содержимое и избегать повторений. Это также означает, что как автор я стараюсь не комментировать многие другие книги о Python, которые могут оказаться для вас весьма полезными и в которых могут обсуждаться темы, не рассматривающиеся ни в одной из моих книг. Упоминания об этих книгах вы найдете в Интернете или в других источниках информации о Python. Все три мои книги отражают опыт 13 лет преподавания Python и путь, пройденный от момента выхода первого издания книги «Programming Python» в 1995 году. (Здесь вполне уместно поместить фотографию убеленного сединами путешественника-исследователя.)

Конечно, некоторые приемы языка демонстрируются в примерах этой книги, а в крупных примерах иллюстрируется, как можно объединять базовые концепции в действующие программы. Особенности объектно-ориентированного программирования, например, часто лучше всего демонстрировать на примерах крупных программ, которые и будут представлены здесь. Однако формально эта книга предполагает, что вы уже знакомы с основами языка Python в достаточной степени, чтобы понять примеры программного кода. Основное внимание здесь будет уделяться библиотекам и инструментам. Поэтому если программный код, который будет демонстрироваться, покажется вам непонятным, – справляйтесь в других источниках.

Второй из этих двух пунктов отражает тот факт, что за многие годы об этой книге сложились неверные представления (вероятно, стоило назвать эту книгу «Применение Python», будь мы чуть более прозорливы в далеком 1995 году). Я хочу ясно обозначить: эта книга – *не* справочник. Это *учебник*. Здесь вы можете найти некоторые подробности, воспользовавшись оглавлением или алфавитным указателем, но данная книга не предназначалась для использования именно в таких целях. Краткий справочник вы найдете в книге под названием «Python Pocket Reference», который вы найдете весьма полезным, как только начнете самостоятельно писать нетривиальный программный код. Существуют и другие источники справочной информации, в том числе другие книги и собственный набор справочных руководств по языку Python. Цель этой книги – постепенное обучение применению языка Python для решения типичных задач, а не подробное описание мельчайших особенностей.

О четвертом издании

Если это первое издание книги, которое вы видите, последние изменения, вероятно, вас будут интересовать меньше всего, поэтому вы можете просто перейти к следующему разделу. Для тех, кто читал предыдущие издания, можно отметить, что четвертое издание этой книги содержит три важных изменения:

- Оно охватывает только Python 3.X.
- Оно было сокращено, чтобы сделать книгу еще направленнее и освободить место для новых тем.
- В него было добавлено обсуждение новых тем и инструментов, появившихся в мире Python.

Первый из этих пунктов является, пожалуй, наиболее важным – это издание опирается на Python 3.X, на стандартную библиотеку для этой версии и на распространенные приемы программирования, используемые его пользователями. Однако чтобы объяснить, как это и два других изменения отразились на данном издании, я должен поведать о некоторых деталях.

Изменения в этом издании

Предыдущие версии книги получили весьма широкое распространение, поэтому ниже я приведу некоторые из наиболее заметных изменений в этом издании:

Существовавший ранее материал был сжат, чтобы освободить место для новых тем

Предыдущее издание книги также имело объем около 1600 страниц, что не позволило выделить достаточно места для рассмотрения новых тем (одна только ориентированность Python 3.X на использование Юникода предполагает массу нового материала). К счастью, недавние изменения в мире Python позволили нам без особого ущерба выкинуть часть существующего материала и освободить место для новых тем.

Глубина обсуждения оставшихся тем при этом не пострадала – эта книга остается такой же основательной, как и прежде. Тем не менее одной из основных задач, стоявших при подготовке этого издания, было не допустить дальнейшего роста его объема, а множество других изменений и сокращений, о которых я упомяну ниже, были сделаны отчасти для того, чтобы включить новые темы.

Рассматривается только Python 3.X

Примеры и пояснения были изменены с учетом того, что теперь эта книга охватывает только версию Python 3.X. Версия Python 2.X больше не поддерживается, за исключением тех особенностей, которые перекочевали из версии 2.X в версию 3.X без изменений. Несмотря на то, что таких особенностей достаточно много, благодаря чему читатели могут использовать версию 2.X, тем не менее формально книга поддерживает только версию 3.X.

В свою очередь это обстоятельство явилось основным фактором, обеспечившим сохранение объема этого издания на прежнем уровне. Ограничившись поддержкой только версии Python 3.X – несовместимой с версией Python 2.X, которую следует рассматривать как будущее языка Python, – нам удалось избежать дублирования описания особенностей, отличающихся в этих двух версиях Python. Такое ограничение поддерживаемых версий особенно важно для такой книги, как эта, где приводится множество расширенных примеров, так как это позволяет демонстрировать примеры только для одной версии.

Для тех, кто по-прежнему пытается удержаться в обоих мирах, 2.X и 3.X, я подробнее расскажу об изменениях в Python 3.X ниже, в этом же предисловии. Самым важным, пожалуй, изменением в версии 3.X, из тех, что описываются в книге, является усовершенствованная поддержка интернационализации в примерах программ PyEdit и PyMailGUI. Несмотря на то, что в версии 2.X также имеется поддержка Юникода, тем не менее усовершенствованная ее реа-

лизация в версии 3.X вынуждает заново пересмотреть реализацию подобных систем, прежде ориентированных на работу с кодировкой ASCII.

Включение недавно появившихся библиотек и инструментов

С момента выхода предыдущего издания появилось или получило дальнейшее развитие множество новых библиотек и инструментов, и они также упоминаются здесь. В их число входят новые инструменты стандартной библиотеки языка Python, такие как модули `subprocess` (рассматривается в главах 2 и 3) и `multiprocessing` (рассматривается в главе 5), а также новые веб-фреймворки, созданные сторонними разработчиками, и инструменты ORM (Object-Relational Mapping – объектно-реляционное отображение) для работы с базами данных. Большинство из них рассматриваются не очень подробно (многие популярные расширения сами по себе являются сложными системами и гораздо подробнее описаны в соответствующей литературе), но для них дается по крайней мере краткое описание в виде резюме.

Например, новая библиотека виджетов Tk `tkinter.ttk` рассматривается в главе 7, но весьма кратко. Как правило, в этом издании мы предпочитали упоминать подобные расширения по ходу дела, вместо того чтобы представлять примеры программного кода без внятного пояснения.

Это предисловие было ужато

Я удалил все инструкции по запуску и использованию примеров программ. Поэтому теперь за инструкциями по использованию обращайтесь к файлам `README`, входящим в состав дистрибутива с комплектом примеров. Кроме того, я убрал большую часть благодарностей, потому что они повторяют благодарности из книги «Изучаем Python», – так как знакомство с книгой «Изучаем Python» теперь считается необходимой предпосылкой, дублирование одного и того же материала здесь ничем не оправдано. Кроме того, было убрано описание содержимого книги – чтобы ознакомиться со структурой книги, обращайтесь к оглавлению.

Была убрана вводная глава с обзором языка Python

Я удалил «организаторскую» главу, присутствовавшую в предыдущем издании, где описывались сильные стороны языка Python, представлялись наиболее видные пользователи, рассматривались различные философии и так далее. Обращение в свою веру играет важную роль в любой сфере, где вопрос «почему» задается менее часто, чем должен бы. В действительности, если бы мастера Python не занимались его популяризацией, все мы, вероятно, использовали бы сейчас Perl или языки командных оболочек!

Однако присутствие здесь такой главы стало совершенно излишним из-за наличия сходной главы в книге «Изучаем Python». Поскольку

книга «Изучаем Python» должна предшествовать этой книге, я решил не расходовать книжное пространство на повторную агитацию «Питонистов» («Pythonista»). В этой книге предполагается, что вы уже знаете, почему стоит использовать Python, поэтому мы сразу же перейдем к его применению.

Было убрано заключительное послесловие

Заключительное послесловие к этой книге было написано еще для первого издания, и теперь ему исполнилось уже 15 лет. Естественно, что оно отражает взгляды на Python, в большей степени характерные для того времени. Например, использование языка Python для разработки гибридных приложений казалось более значимым в 1995 году, чем в 2010. В современном, более обширном мире Python большинству пользователей не приходится иметь дело со связанным программным кодом на языке C.

В предыдущих изданиях я добавлял новое послесловие для каждого издания, чтобы уточнить и дополнить идеи, представленные в заключении к книге. Теперь я убрал эти послесловия, заменив их коротким примечанием. Само заключение я решил оставить, потому что оно по-прежнему актуально для многих читателей и имеет некоторую историческую ценность. Да к тому же удачные шутки...

Было убрано вступительное слово

По похожим причинам, представленным в двух предыдущих пунктах, я убрал разделы со вступительным словом к предыдущим изданиям. Те, кому это будет интересно, историческую справку о вкладе Гвидо ван Россума (Guido van Rossum), создателя Python, в развитие языка смогут найти в Интернете. Если вам интересно, как за эти годы изменился язык Python с технической точки зрения, смотрите документ «What's New» («Что нового»), входящий в состав стандартного комплекта руководств по языку Python (доступен по адресу *http://www.python.org/doc* и устанавливается вместе с Python в Windows и на других платформах).

Раздел, посвященный интеграции с языком C, был сокращен до одной главы

Я сократил раздел, посвященный созданию расширений на языке C и встраиванию сценариев на языке Python в программы на языке C, до одной короткой главы в конце части, посвященной инструментальным средствам, где коротко описываются основные понятия, связанные с этой темой. Проблемы связывания программ на языке Python с библиотеками на языке C на сегодняшний день волнуют лишь незначительную часть пользователей, а те, кому эти навыки действительно необходимы, найдут более полный пример интеграции в исходных текстах самого языка Python. Имеющиеся возможности перечислены здесь достаточно полно, но значительный объем программного кода на языке C был вырезан в расчете на то, что более

представительные примеры вы найдете в программном коде реализации самого языка Python.

Часть, посвященная системному программированию, была сокращена и переработана

Прежние две главы с большими примерами использования Python в системном программировании были объединены в одну, более короткую главу с новыми или значительно переработанными примерами. Фактически эта часть (часть II) подверглась самым значительным изменениям. Она включает описание новых инструментов, таких как модули `subprocess` и `multiprocessing`, знакомит читателей с сокетами, а кроме того, из нее были удалены устаревшие сведения и примеры, унаследованные из прежних изданий. Честно признаться, несколько примеров работы с файлами были созданы еще в 1990-х годах и оказались сильно устаревшими. Начальная глава в этой части была разбита на две, чтобы упростить чтение материала (описание контекста командной оболочки, включая потоки ввода-вывода, было вынесено в отдельную главу), а несколько листингов крупных программ (включая запускающие сценарии с автоматической настройкой) теперь вынесены за пределы книги и включены в состав дистрибутива с комплектом примеров.

Некоторые крупные примеры были исключены из книги (но остались в составе дистрибутива с комплектом примеров)

Точно так же из книги были исключены два крупных примера, демонстрирующих создание графического интерфейса, *PyTree* и *PyForm*. Однако их обновленная реализация доступна в комплекте примеров. В этом издании вы все еще сможете найти упоминание и описание множества крупных примеров, включая примеры реализации полноценных клиентов электронной почты с графическим и веб-интерфейсом, а также программы для просмотра изображений, калькуляторы, часы, текстовые редакторы с поддержкой Юникода, простые графические редакторы, сценарии регрессивного тестирования и многие другие. Однако, так как программный код примеров не добавляет ничего важного к раскрытию темы и вообще эти примеры предлагались в основном для самостоятельного изучения, – я перевел их в разряд дополнительной информации и исключил их из текста этого издания книги.

Обширная глава с описанием тем, касающихся Интернета, была заменена кратким обзором

Я полностью убрал обширную главу с описанием тем, касающихся Интернета, оставив лишь краткий обзор в начале части (с акцентом на возможностях создания графического интерфейса, который приводится в начале третьей части «Программирование GUI»). Здесь вы найдете все ранее включавшиеся в рассмотрение инструменты, такие как веб-фреймворк ZOPE, объектная модель COM, технологии Windows Active Scripting и ASP, HTMLgen, Python Server Pages

(PSP), Jython и уже серьезно устаревшая система Grail. Некоторые из этих инструментов по-прежнему заслуживают положительных оценок, но в этом издании никакие из них не рассматриваются подробно. В обзор были добавлены новые инструменты (включая многие из тех, что перечислены в следующем абзаце), но, опять же, весьма краткие и без примеров программного кода.

Несмотря на все попытки автора угадать направления развития веб-технологий в будущем, печатное издание не способно полностью соответствовать эволюции сферы развития Интернета. Например, в настоящее время появились веб-фреймворки, такие как Django, Google App Engine, TurboGears, pylons и web2py, соперничающие по своей популярности с ZOPE. Точно так же фреймворк .NET Framework во многих приложениях вытеснил объектную модель Windows COM. Реализация IronPython теперь способна обеспечить такую же тесную интеграцию с .NET, как и Jython с Java. А механизм Active Scripting в значительной степени может быть замещен клиентскими фреймворками, основанными на JavaScript и использующими технологию AJAX, такими как Flex, Silverlight и pyjamas (которые часто называют средствами разработки полнофункциональных веб-приложений). Кроме собственно исторической ценности, примеры, ранее представленные в этой категории, не давали возможности изучить описываемые инструменты или хотя бы судить об их достоинствах.

Вместо того чтобы включать в книгу неполное (и практически бесполезное) описание инструментов, которые в течение предполагаемого срока жизни этого издания могут как уйти далеко вперед в своем развитии, так и исчезнуть со сцены, я решил предоставить лишь краткие обзоры наиболее важных тем, касающихся Интернета, и предлагаю читателям самим постараться найти необходимые подробности в Интернете. По сути, основная цель книги, которую вы читаете, состоит в том, чтобы дать углубленные знания об основах Интернета, которые позволят вам воспользоваться более сложными системами, когда вы будете готовы сделать шаг вперед.

Единственное исключение: описание XML, присутствовавшее ранее в этой главе, было дополнено и перемещено в главу, посвященную обработке текста (где оно и должно было бы находиться). Точно так же было сохранено описание объектно-ориентированной базы данных ZODB, поддерживаемой фреймворком ZOPE, хотя и в сильно урезанном виде, чтобы получить возможность добавить описание механизмов ORM, таких как SQLObject и SQLAlchemy (также в краткой форме).

Задействованы современные инструменты, доступные в версии 3.X

К моменту написания этих строк Python 3.X все еще находился на этапе внедрения, и некоторые из инструментов сторонних разработчиков, которые использовались в примерах в предыдущих из-

даниях этой книги, по-прежнему доступны только в версиях для Python 2.X. Чтобы обойти этот временный недостаток, я изменил некоторые примеры, задействовав в них альтернативные инструменты, обеспечивающие поддержку версии Python 3.X.

Наиболее заметным в этом смысле является раздел, посвященный базам данных SQL, – теперь в нем вместо интерфейса доступа к серверу MySQL, присутствующего в Python 2.X, используется библиотека SQLite поддержки баз данных, встраиваемых в приложения, которая в версии 3.X стала стандартной частью Python. К счастью, переносимый прикладной интерфейс Python позволяет сценариям взаимодействовать с обоими механизмами практически одинаково, поэтому такое изменение является весьма незначительной жертвой.

Отдельно следует отметить использование расширения PIL для отображения изображений в формате JPEG в части, посвященной созданию графического интерфейса. Это расширение было адаптировано Фредриком Лундом (Fredrik Lundh) для версии Python 3.1 как раз к моменту подготовки этого издания. Когда я сдавал в издательство окончательный вариант рукописи этой книги в июле 2010 года, эта версия расширения официально еще не была выпущена, но она должна была вскоре выйти; поэтому в качестве временной меры заплаты для этой библиотеки, обеспечивающие поддержку Python 3.X, были включены в комплект примеров.

Было исключено описание дополнительных возможностей языка

Все дополнительные особенности языка Python, такие как дескрипторы, свойства, декораторы, метаклассы и поддержка Юникода, являются частью языка Python. Поэтому их описание было перемещено в четвертое издание книги «Изучаем Python». Например, улучшенная поддержка Юникода и ее влияние на приемы работы с файлами, именами файлов, сокетами и со многими другими инструментами обсуждаются в этой книге, но основы Юникода здесь не рассматриваются. Некоторые темы из этой категории определенно имеют прикладной характер (или, по крайней мере, представляют интерес для разработчиков инструментальных средств и архитекторов прикладных интерфейсов), но наличие их описания в книге «Изучаем Python» позволило избежать дальнейшего увеличения объема этой книги. Ищите подробное обсуждение этих тем в книге «Изучаем Python».

Прочие незначительные изменения

Естественно, что попутно было внесено множество мелких изменений. Например, для размещения элементов форм теперь вместо метода `pack` используется метод `grid` из библиотеки `tkinter`, потому что он обеспечивает более непротиворечивый способ размещения элементов в платформах, где размер шрифта в подписях не соответствует высоте полей ввода (включая ОС Windows 7 netbook, установленную на ноутбуке, использовавшемся для работы над этим изда-

нием). Кроме того, по всей книге были добавлены новые сведения, включая новое описание механизма переадресации потоков ввода-вывода в сокеты, в части с описанием приемов работы в Интернете; новый многопоточный диалог поиска по регулярным выражениям с поддержкой Юникода и изменения в тестах для примера *PyEdit*; множество других изменений, которые, вероятно, вам будет лучше раскрывать по ходу дела, чем читать о них в предисловии.

Наконец, некоторые блоки комментариев, начинающиеся с символа «#» и расположенные в начале файлов с исходными текстами, я заменил строками документирования (и, для единообразия, даже в сценариях, которые не предназначены для импортирования, хотя отдельные строки «#» остались в крупных примерах, где они отделяют текст). Я также заменил несколько устаревших операторов «while 1» на «while True»; чаще стал использовать оператор +=; внес другие изменения, исправив некоторые устаревшие шаблоны программирования. Старые привычки бывает сложно искоренять, но подобные изменения делают примеры не только более функциональными, но и более полно отражающими современные приемы программирования.

Несмотря на добавление новых тем, в общей сложности было удалено четыре главы (нетехническое введение, одна из глав с примерами по системному программированию, глава с расширенными темами, касающимися Интернета, и одна объединительная глава). Были урезаны несколько дополнительных примеров и сопутствующий им материал (включая PyForm и PyTree), а также, специально для экономии пространства, книга ограничивается представлением только версии Python 3.X и описанием лишь самых основ разработки приложений.

Что же осталось?

В результате изменений, обозначенных выше, это издание получилось более кратким и более четко отражающим основную его роль – учебное руководство по применению языка Python для решения типичных задач программирования. Однако, учитывая объем книги, можно смело утверждать, что это по-прежнему обстоятельная и *всесторонняя* книга, предназначенная быть первой ступенью на пути овладения мастерством разработки приложений на языке Python.

Вопреки последним настроениям (и рискуя получить клеймо еретика) я совершенно уверен, что книги, подобные этой, должны подтягивать своих читателей вверх, а не опускаться до их уровня. Снижение интеллектуальной планки вредит не только читателям, но и той области знаний, в которой они предполагают действовать. В этой книге вы не найдете множество забавных рисунков, как в некоторых, и она не будет поражать вас развлекательным повествованием вместо технической глубины. Цель моих книг состоит в том, чтобы передать сложные поня-

тия убедительным и надежным способом и снабдить вас инструментами, которые потребуются вам в разработке программного обеспечения.

Разумеется, существует множество типов обучающихся, и ни одна книга не сможет работать на любую аудиторию. Фактически именно по этой причине изначальная версия этой книги позднее была разделена на две, и описание основ языка было делегировано отдельной книге «Изучаем Python». Кроме того, и *программистов* можно поделить на две категории – тех, кто желает получить глубокие знания в области разработки программного обеспечения, и *скриптеров*, не испытывающих такой потребности. Некоторым вполне достаточно иметь элементарные знания, позволяющие дорабатывать системы или библиотеки и решать текущие проблемы. Но это пока они не начнут вторгаться в область разработки полномасштабных приложений – порог, за которым в худшем случае может наступить разочарование, а в лучшем – лучшее понимание сложной природы этой области.

Неважно, к какому лагерю вы принадлежите, важно понять основное назначение этой книги. Если вы ищете кратчайший путь к мастерству, вам едва ли понравится эта книга (как и разработка программного обеспечения в целом). Однако если вы стремитесь научиться хорошо программировать на языке Python и одновременно получить удовольствие от процесса изучения, эта книга наверняка станет для вас важной вехой на пути обретения опыта.

В конечном счете, овладеть навыками программирования далеко не так просто, как пытаются представить некоторые. Однако если вы приложите необходимые усилия, то обнаружите, что это стоит затраченного времени. Это особенно верно для тех, кто вооружает себя удобным инструментом программирования, таким как язык Python. Ни одна книга и ни одни курсы не превратят вас в «Повелителя Вселенной» Python, тем не менее цель этой книги состоит в том, чтобы помочь вам добиться этого, сократив начальный этап освоения и обеспечив надежный фундамент в наиболее типичных областях применения Python.

Влияние Python 3.X на эту книгу

Как уже упоминалось выше, это издание теперь охватывает только версию Python 3.X. Версия языка Python 3.X несовместима с версией 2.X. В своей основе язык версии 3.X очень напоминает Python 2.X, но имеет множество существенных отличий, кроющихся как в самом языке, так и в стандартной библиотеке. Читатели, не имеющие опыта работы с версией 2.X, могут пропустить описание этих отличий, но появившиеся изменения очень сильно повлияли на содержание этого издания. Для широкого круга пользователей Python 2.X в этом разделе описываются наиболее значимые изменения, относящиеся к этой категории.

Если вам интересно поближе познакомиться с отличиями от версии 2.X, я предлагаю дополнительно найти четвертое издание книги

«Python Pocket Reference», упомянутое выше. Там приводятся описания основных структур языка обеих версий, 2.X и 3.X, встроенных функций и исключений, а также перечисляется большинство модулей из стандартной библиотеки и инструментов, используемых в этой книге. Хотя четвертое издание книги «Изучаем Python» не является справочником по различиям между версиями, оно охватывает обе версии, 2.X и 3.X, и, как уже отмечалось, знакомство с ней является обязательным условием для усвоения материала этой книги. Цель этого издания книги «Программирование на Python», ориентированного только на версию 3.X, заключается вовсе не в том, чтобы оставить за бортом огромное количество пользователей версии 2.X, а в том, чтобы помочь читателям перейти на новую версию и избежать увеличения размеров и без того массивной книги.

Изменения, связанные с включением поддержки версии 3.X

К счастью, многие отличия между версиями 2.X и 3.X, обусловившие необходимость внесения изменений в эту книгу, достаточно тривиальны. Например, библиотека `tkinter`, широко используемая в этой книге для построения графических интерфейсов, присутствует в версии 3.X под именем `tkinter` и имеет структуру пакета – ее прежняя инкарнация в версии 2.X, в виде модуля `Tkinter`, в этой книге не описывается. Это отличие приводит, в основном, к необходимости использовать отличные инструкции импортирования, но здесь приводятся только инструкции для версии Python 3. Аналогично с целью соблюдения соглашений об именовании модулей, принятых в версии 3.X, модули для версии 2.X `anydbm`, `Queue`, `thread`, `StringIO.StringIO` и `urllib.open` превратились в Python 3.X и в этом издании в модули `dbm`, `queue`, `_thread`, `io.StringIO` и `urllib.request.urlopen` соответственно. Точно так же были переименованы и другие инструменты.

С другой стороны, переход к версии 3.X предполагает более широкие идиоматические изменения, которые, конечно же, являются более радикальными. Например, усовершенствованная поддержка Юникода в Python 3.X подтолкнула к созданию для этого издания примеров полностью интернационализированных версий текстового редактора PyEdit и клиента электронной почты PyMailGUI (подробнее об этом чуть ниже). Кроме того, замена модуля `os.popen2` модулем `subprocess` потребовала включения новых примеров; отказ от модуля `os.path.walk` в пользу модуля `os.walk` позволил сократить некоторые примеры; новое разделение на файлы и строки Юникода и двоичные файлы и строки потребовало изменения группы дополнительных примеров и описания; кроме того, появились новые модули, такие как `multiprocessing`, предлагающие новые возможности, которые необходимо было описать в этом издании.

Помимо изменений в библиотеке, в примерах этого издания также отражены изменения в языке Python 3. Например, здесь учтены все изменения, коснувшиеся функций из версии 2.X `print`, `raw_input`, `keys`, `has_key`, `map` и `apply`. Кроме того, новая модель *импортирования относительно пакетов*, появившаяся в версии 3.X, нашла отражение в некоторых примерах, таких как `mailtools` и анализаторы выражений, а отличия в поведении *операторов деления* вынудили внести небольшие изменения в примеры создания графического интерфейса, такие как PyClock, PyDraw и PyPhoto.

Замечу также, что я не стал заменять все выражения форматирования строк на основе оператора `%` новым методом `str.format`, потому что оба способа форматирования поддерживаются в Python 3.1, и похоже, оба они будут поддерживаться еще очень долго, если не всегда. Фактически если воспользоваться поиском с помощью регулярных выражений, который мы реализуем в примере текстового редактора PyEdit в главе 11, можно обнаружить, что этот оператор встречается более 3000 раз в программном коде библиотеки для Python 3.1. Я не могу с абсолютной точностью предсказать, как будет развиваться Python в будущем, поэтому обращайтесь к первой главе, где подробнее рассказывается об этом, если когда-нибудь вам потребуется внести изменения.

Кроме того, из-за того, что это издание охватывает только версию 3.X, оказалось невозможным использовать некоторые сторонние пакеты, существующие только в версии для Python 2.X, о чем уже говорилось выше. В их число входят интерфейс к MySQL, ZODB, PyCrypto и другие. Как уже упоминалось выше, для работы под управлением Python 3.1 была адаптирована библиотека PIL с целью использования в этой книге, но для этого потребовалось наложить специальные исправления, а официальная версия, поддерживающая Python 3.X, до сих пор не вышла. Многие из недостающих модулей для версии 3.X могут появиться к моменту, когда вы будете читать эти строки, либо в виде адаптированных версий для Python 2.X, либо в виде совершенно новых версий, специально для Python 3.X.

Особенности языка и библиотека: Юникод

Поскольку эта книга посвящена изучению принципов разработки приложений, а не основ языка программирования, изменения в языке не обязательно должны отслеживаться здесь. В действительности, оглядываясь на книгу «Изучаем Python», можно сказать, что изменения в языке, связанные с переходом на версию 3.X скорее касаются ее, а не данной книги. В большинстве случаев изменения в примерах к этой книге были обусловлены необходимостью сделать их более понятными или более функциональными, а не включением поддержки версии 3.X.

С другой стороны, переход на версию Python 3.X оказывает влияние на значительную часть программного кода, и иногда это влияние может оказаться весьма тонким. Тем не менее читатели с опытом использова-

ния Python 2.X обнаружат, что если отличия в версии 3.X языка чаще всего легко преодолимы, то отличия в стандартной библиотеке для версии 3.X преодолеть иногда оказывается гораздо сложнее.

Но к наиболее широким последствиям привело главное изменение в Python 3.X – улучшенная поддержка строк Юникода. Будем честными: поддержка Юникода в версии 3.X иногда может существенно осложнять жизнь тех, кто всю жизнь сталкивался только с кодировкой ASCII! Как мы увидим далее в этой книге, эта поддержка оказывает существенное влияние при работе с содержимым файлов, с именами файлов, с дескрипторами каналов, с сокетами, при выводе текста в графическом интерфейсе, при работе с такими протоколами Интернета, как FTP и email, при разработке сценариев CGI и даже при использовании некоторых инструментов хранения данных. Плохо это или хорошо, но как только мы войдем в мир разработки приложений, описываемый в этой книге, Юникод перестанет быть необязательной темой для многих, если не для большинства программистов на языке Python 3.X.

Конечно, если уж на то пошло, никому и никогда не следовало бы рассматривать использование Юникода, как дополнительную и необязательную возможность. Далее мы увидим, что некоторые приемы, которые, казалось бы, работают в Python 2.X, на самом деле нельзя признать удовлетворительными – работа с текстом как с простым набором байтов может порождать различные проблемы, такие как ошибки при сравнении строк в различных кодировках (утилита grep, реализованная в составе текстового редактора PyEdit в главе 11, является ярким примером программного кода, который должен терпеть неудачу при отказе от использования Юникода). Python 3.X обнажает подобные проблемы, делая их более заметными для программиста.

Однако перенос нетривиального программного кода на версию 3.X не является неразрешимой задачей. Кроме того, многие читатели этого издания имеют шикарную возможность начать использовать язык Python сразу же с версии 3.X и не иметь дела с существующим программным кодом для версии 2.X. Если вы принадлежите к их числу, вы увидите, что Python 3.X является надежным языком широкого применения для разработки сценариев и приложений, стремящимся устранить многие проблемы, которые когда-то скрывались в версии 2.X.

Ограничения Python 3.1: электронная почта, CGI

Здесь необходимо сделать одно важное замечание, касающееся основных примеров в книге. Чтобы их реализация представляла интерес для как можно более широкой аудитории, основные примеры в этой книге имеют отношение к *электронной почте* и обеспечивают поддержку интернационализации и Юникода. К этой категории относятся примеры PyMailGUI (в главе 14) и PyMailCGI (в главе 16), а также все предшествующие примеры, используемые этими приложениями, в число кото-

рых входит текстовый редактор PyEdit, поддерживающий Юникод при работе с файлами, при отображении и поиске текста.

В этом есть свои плюсы и минусы. Плюс в том, что в конечном итоге мы получим полнофункциональный и интернационализированный клиент электронной почты PyMailGUI, использующий существующий ныне пакет `email`. Это приложение будет поддерживать любые кодировки для содержимого и заголовков электронных писем и обеспечит возможность не только их просмотра, но и составления. Минус заключается в том, что при этом придется приложить определенные усилия, чтобы обойти сложности, связанные с особенностями реализации пакета `email` в Python 3.1.

К сожалению, как будет показано в главе 13, в пакете `email` в Python 3.1 имеется ряд проблем, связанных с обработкой типов `str/bytes` в Python 3.X. Например, отсутствует простой способ определить кодировку для преобразования текста письма, возвращаемого модулем `poplib` в виде объекта типа `bytes`, в тип `str`, который ожидает получить парсер `email`. Кроме того, в настоящее время пакет `email` не в состоянии обрабатывать некоторые виды сообщений, а поддержка некоторых типов сообщений реализована неправильно или слишком специфично.

Эта ситуация носит, скорее всего, временный характер. Исправление некоторых из проблем, с которыми нам придется столкнуться в этой книге, уже запланировано разработчиками. (Фактически из-за одного из таких исправлений, выполненных в версии 3.2, в последний момент потребовалось внести изменения в один из примеров в главе 13.) Кроме того, ведется разработка новой версии пакета `email`, в которой будут учтены все особенности поддержки Юникода и типа `bytes` в версии 3.X, но новая версия пакета будет выпущена значительно позже выхода этой книги, и она может оказаться несовместимой с API текущей версии пакета, как и сам Python 3.X. По этой причине в книге приводятся не только обходные решения, но и попутно делаются некоторые предположения. Настоятельно рекомендую регулярно посещать веб-сайт книги (описывается ниже), где будут приводиться сведения об изменениях в будущих версиях Python. Один из плюсов этой ситуации состоит в том, что детальное описание проблемы отражает существующие реалии разработки приложений – основной темы этой книги.

Проблемы, наблюдающиеся в пакете `email`, также в значительной степени были унаследованы реализацией модуля `cgi` в версии 3.1. Сценарии CGI – это одна из простейших технологий, на смену которой приходят веб-фреймворки, тем не менее они по-прежнему могут служить целям обучения основам Веб и все еще составляют основу множества крупных наборов инструментальных средств. Скорее всего, эти недостатки версии 3.1 также будут исправлены в будущем, но нам придется приложить определенные усилия, чтобы реализовать в сценариях CGI выгрузку текстовых файлов в главах 15 и 16 и вложений в сообщениях электронной почты в приложении PyMailCGI. Казалось бы, спустя два

года после выхода версии Python 3.0 такое положение дел как минимум недопустимо, но такова жизнь в динамическом мире разработки программного обеспечения и в мире книг, которые стремятся вести за собой, а не плестись в хвосте.

Использование примеров из книги

Поскольку примеры составляют основную часть содержимого этой книги, я должен сказать несколько слов о них.

Где искать примеры и обновления

Как и прежде, примеры, обновления, исправления и дополнения к этой книге можно найти на веб-сайте автора, по адресу:

http://www.rmi.net/~lutz/about-pp4e.html

На этой странице веб-сайта моей книги будут даваться ссылки на дополнительную информацию, имеющую отношение к данной версии книги. Однако я не являюсь владельцем этого доменного имени, поэтому если указанная ссылка перестанет работать на протяжении срока жизни этого издания, попробуйте обратиться на другой сайт, по адресу:

http://learning-python.com/books/about-pp4e.html
(альтернативная страница)

Если перестанут действовать обе ссылки, попробуйте выполнить поиск в Интернете (не сомневаюсь, что большинство читателей предпримут именно этот шаг).

На веб-сайте книги (а также на сайте издательства O'Reilly, о котором говорится в следующем разделе), где бы он ни находился, вы сможете загрузить *дистрибутив с комплектом примеров* – файл архива, содержащий все примеры, которые приводятся в книге, а также несколько дополнительных примеров, которые упоминаются, но отсутствуют в самой книге. Чтобы иметь возможность опробовать примеры и избавить себя от необходимости вводить их вручную, загрузите архив, распакуйте его и ознакомьтесь с содержимым файла *README.txt*, где приводятся инструкции по использованию. Порядок именования файлов примеров и структуру дерева каталогов пакета я опишу ниже, когда мы перейдем к опробованию первого сценария в первой главе.

Как и в случае с первыми тремя изданиями, я буду поддерживать на этом веб-сайте неофициальный «блог», где будут описываться изменения в языке Python, а также будут даваться пояснения и обновления в книге, который вы можете рассматривать, как дополнительное приложение.

Кроме того, на веб-сайте издательства O'Reilly, о котором говорится ниже, имеется система регистрации сообщений об опечатках, где вы сможете сообщить об обнаруженных ошибках. Аналогичная страни-

ца присутствует на моем сайте книги. Я стараюсь обновлять веб-сайты моих книг как можно чаще, тем не менее может так получиться, что страница на сайте издательства O'Reilly со списком опечаток будет содержать более свежую информацию. В любом случае в качестве официального источника информации об ошибках и исправлениях следует рассматривать объединение этих двух списков.

Переносимость примеров

Примеры для этой книги разрабатывались, тестировались и запускались под управлением ОС Windows 7 и Python 3.1. Непосредственно перед передачей книги в печать все основные примеры успешно прошли тестирование под управлением грядущей версии Python 3.2 (третья альфа-версия), то есть все, о чем рассказывается в этой книге, в равной степени относится и к Python 3.2. Кроме того, программный код на языке C из главы 20 и ряд примеров параллельного программирования были опробованы в Windows под управлением оболочки Cygwin, имитирующей окружение Unix.

Несмотря на то, что Python и стандартная библиотека, вообще говоря, нейтральны в отношении используемой платформы, тем не менее в некоторые примеры необходимо будет внести незначительные изменения, чтобы их можно было опробовать на других платформах, таких как Mac OS X, Linux и в других разновидностях ОС Unix. Примеры с графическим интерфейсом на основе пакета `tkinter`, а также некоторые примеры из раздела, посвященного системному программированию, могут быть особенно чувствительны к различиям между платформами. Часть проблем, связанных с переносимостью, будут отмечаться в ходе обсуждения примеров, но некоторые проблемы могут не упоминаться явно.

Так как у меня не было ни времени, ни возможности протестировать примеры на всех возможных платформах, какие только могут иметься в распоряжении читателей, адаптацию этих примеров для конкретной платформы предлагается рассматривать, как упражнения для самостоятельного решения. Если вы обнаружите какую-либо зависимость от типа платформы и пожелаете поделиться своими исправлениями, обращайтесь на мой сайт книги, указанный выше, – я буду рад донести до других читателей ваши исправления для любых платформ.

Сценарии запуска демонстрационных программ

В состав пакета с примерами, который был описан выше, также входят сценарии PyDemos и PyGadgets запуска демонстрационных программ. Они позволяют быстро ознакомиться с некоторыми основными примерами с графическим и веб-интерфейсом. Запускающие их сценарии, находящиеся на верхнем уровне дерева каталогов пакета с примерами, предназначены для настройки пути поиска модулей в запускаемых ими программах и могут использоваться непосредственно на совместимых

платформах, включая Windows. Более подробную информацию об этих сценариях вы найдете в файлах README, а также в кратких обзорах, которые приводятся в конце главы 6 и 10.

Политика повторного использования программного кода

Мы прерываем предисловие, чтобы вставить несколько слов от имени юридического отдела. Данная книга призвана оказать вам помощь в решении ваших задач. Вообще вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам *необходимо* получить разрешение от издательства O'Reilly. Для цитирования данной книги или примеров из нее, при ответе на вопросы не требуется получение разрешения. При включении существенных объемов программного кода примеров из этой книги в вашу документацию вам *необходимо* будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание названия книги, авторов, издательства и ISBN. Например: «Programming Python, Fourth Edition, by Mark Lutz (O'Reilly). Copyright 2011 Mark Lutz, 978-0-596-15810-1».

Как связаться с издательством O'Reilly

В предыдущем разделе я описал свои собственные сайты, где можно найти примеры и обновления. В дополнение к тем сайтам вы можете обратиться на сайт издательства с вопросами и предложениями, касающимися этой книги:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный)
707-829-0104 (факс)

Как уже говорилось выше, на сайте издательства O'Reilly поддерживается веб-страница для этой книги, где можно найти список опечаток, файлы с примерами и другую дополнительную информацию:

http://www.oreilly.com/catalog/9780596158101

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, программное обеспечение, Центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Типографские соглашения

В этой книге приняты следующие соглашения:

Курсив

Курсив применяется для выделения имен файлов и каталогов, новых терминов и некоторых комментариев в примерах программного кода.

`Моноширинный шрифт`

Применяется для представления листингов программ, а также для выделения в обычном тексте программных элементов, таких как имена модулей, методов, параметров, классов, функций, инструкций, программ, объектов и тегов HTML.

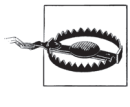
`Моноширинный жирный`

Используется для выделения команд или текста, который должен быть введен пользователем.

`Моноширинный курсив`

Обозначает элементы в программном коде, которые должны быть замещены, исходя из конкретной ситуации.

Так выделяются примечания, имеющие отношение к текущему обсуждению.



Так выделяются предупреждения или предостережения, имеющие отношение к текущему обсуждению.

Благодарности

Я благодарен всем, кто перечислен в предисловии к четвертому изданию книги «Изучаем Python», вышедшему меньше года тому назад. Так как знакомство с книгой «Изучаем Python» является обязательным условием для чтения этой книги, а также потому что люди, помогавшие мне в создании обеих книг, – одни и те же, я не стал повторять весь список здесь. Но, как бы то ни было, я хотел бы выразить благодарность:

- Издательству O'Reilly за продвижение Python и публикацию серьезных и содержательных книг, имеющих отношение к программному обеспечению с открытыми исходными текстами
- Сообществу Python, составляющему большую часть моего мира начиная с 1992 года
- Тысячам студентов, прошедших через 250 курсов обучения языку Python, которые я провел начиная с 1997 года
- Сотням тысяч читателей, прочитавшим 12 изданий всех трех моих книг о Python, которые вышли с 1995 года
- Монти Пайтону (Monty Python), тезке Python, за множество интересных ситуаций, в которых есть чему поучиться (подробнее об этом – в следующей главе)

Книгу пишет обычно один человек, но многие идеи рождаются в сообществе. Я благодарен за отклики, которые мне повезло получить в течение последних 18 лет от моих студентов и читателей. Студенты – лучшие учителя учителей.

С личной стороны я хотел бы сказать спасибо моим братьям и сестре за старые добрые времена, а также моим детям, Майклу (Michael), Саманте (Samantha) и Роксане (Roxanne), за возможность гордиться ими.

А особенная благодарность моей супруге Вере (Vera), которой так или иначе удалось внести немало хорошего в этот в каком-то смысле неизменяемый объект.

Марк Лутц (Mark Lutz), июль 2010

Так что же такое Python?

Как уже говорилось выше, в этой книге не будет уделяться большое внимание основам Python, и мы отложим абстрактные рассуждения о роли Python до заключительной главы, то есть до того момента, когда вы познакомитесь с этим языком программирования на практике. Тем не менее если вы хотите получить исчерпывающее определение темы этой книги, извольте:

Python – это язык программирования общего назначения, распространяемый с открытыми исходными текстами (open source). Он оптимизирован для создания качественного программного обеспечения, высокой производительности труда разработчиков, переносимости программ и интеграции компонентов. Язык Python используется сотнями тысяч разработчиков по всему миру в таких областях, как создание веб-сценариев, системное программирование, создание пользовательских интерфейсов, настройка программных продуктов под пользователя, численное программирование и в других. Как считают многие, один из самых используемых языков программирования в мире.

Как популярный язык, обеспечивающий сокращение времени, затрачиваемого на разработку программ, Python используется для создания широкого круга программ в самых разных областях. В число пользователей Python в настоящее время входят Google, YouTube, Industrial Light & Magic, ESRI, системы BitTorrent обмена файлами, Jet Propulsion Lab в NASA, игра Eve Online и National Weather Service (национальная метеорологическая служба, США). Язык Python используется в самых разных областях, от администрирования систем, разработки веб-сайтов, создания сценариев для мобильных устройств и обучения, до тестирования аппаратуры, анализа капиталовложений, компьютерных игр и управления космическими кораблями.

Среди прочих достоинств, Python отличается удивительной простотой, удобочитаемостью и простым синтаксисом; он легко интегрируется с внешними компонентами, написанными на других языках программирования; имеет мультипарадигменную архитектуру и поддерживает объектно-ориентированное, функциональное и модульное программирование; обладает обширной коллекцией уже запрограммированных интерфейсов и утилит. Набор встроенных инструментальных средств делает его необычайно гибким и динамичным языком программирования, идеально подходящим не только для быстрого решения тактических задач, но и для разработки перспективных стратегических решений. Несмотря на свое общее назначение, Python часто называют *языком сценариев*, так как он позволяет легко и просто использовать другие программные компоненты и управлять ими.

Самым большим достоинством Python является, пожалуй, то, что с его помощью разработка программного обеспечения становится более быстрой и приятной. Есть такая категория людей, для которых программирование является самоцелью. Они наслаждаются самим процессом. Пишут программы исключительно для собственного удовольствия, а коммерческие или карьерные выгоды рассматривают лишь, как вторичное следствие. Именно такие люди в значительной степени причастны к появлению Интернета, движения за распространение программного обеспечения с открытыми исходными текстами (open source) и Python. Эти же люди исторически являются основными читателями этой книги. От них часто можно услышать, что с таким инструментом, как Python, программирование превращается в настоящее развлечение.

Чтобы понять, как это происходит, читайте дальше. Как побочный эффект, значительная часть этой книги представляет собой демонстрацию воплощения идеалов Python в действующий программный код. Как мы увидим далее, особенно когда будем знакомиться с инструментальными средствами, используемыми для создания графического интерфейса пользователя, веб-сайтов, в системном программировании и так далее, Python способствует продвижению новых технологий.

Об авторе

Марк Лутц (Mark Lutz) является ведущим специалистом в области обучения языку программирования Python, автором самых ранних и наиболее популярных публикаций и известен в сообществе пользователей Python своими новаторскими идеями.

Марк является автором книг «Изучаем Python», «Программирование на Python» и «Python Pocket Reference», выпущенных издательством O'Reilly, каждая из которых претерпела уже четыре издания. Он использует Python и занимается его популяризацией начиная с 1992 года; книги о Python начал писать в 1995 году; преподаванием этого языка программирования стал заниматься с 1997 года. На начало 2010 года Марк провел 250 курсов, обучил более 3500 студентов, написал книги по языку Python, суммарный тираж которых составил примерно четверть миллиона копий и которые были переведены более чем на десять языков.

Обладает степенями бакалавра и магистра в области информатики, закончиё университет штата Висконсин (США). На протяжении последних 25 лет занимался разработкой компиляторов, инструментальных средств программиста, приложений и разнообразных систем в архитектуре клиент/сервер. Связаться с Марком можно через веб-сайт книги *http://rmi.net/~lutz* и веб-сайт курсов, которые он ведет: *http://learning-python.com*.

I

Начало

Эта часть книги запускает повествование, предлагая краткий экскурс, в котором рассматриваются фундаментальные понятия языка Python и представляются некоторые из наиболее типичных приемов его использования.

Глава 1

Эта глава начинает рассказ с простого примера – записи информации о людях, – что позволит коротко представить некоторые из основных областей применения языка Python, которые мы будем изучать в этой книге. Мы заставим этот пример существовать в самых разных ситуациях. По пути мы встретимся с базами данных, графическими интерфейсами, веб-сайтами и так далее. Эта своего рода демонстрационная глава задумывалась с целью возбудить в вас интерес. Здесь мы не будем исследовать все аспекты, но у нас будет возможность увидеть Python в действии, прежде чем мы погрузимся в детали. Данная глава служит также обзором некоторых базовых идей языка, с которыми вы должны быть знакомы, прежде чем приступать к чтению этой книги, – такими как представление данных и объектно-ориентированное программирование (ООП).

Назначение этой части книги не в том, чтобы дать вам всесторонний обзор языка Python, а в том, чтобы познакомить вас с примером его применения и предоставить краткий обзор определенного круга задач, решаемых с помощью языка Python.

1

Предварительный обзор

«Программирование на Python»: краткий очерк

Если вы берете в руки книгу такого размера, как эта, то вам, как и большинству людей, перед тем как засучить рукава, наверняка захочется немного узнать о том, что вы собираетесь изучать. Именно об этом рассказывает данная глава – в ней приводятся несколько примеров, которые позволят вам оценить возможности языка Python, прежде чем вы перейдете к изучению подробностей. Здесь вы найдете лишь краткие пояснения, поэтому если у вас появится желание получить подробное описание инструментов и приемов, использованных в этой главе, вам придется прочитать последующие части книги. Цель этой главы состоит в том, чтобы возбудить у вас аппетит кратким обзором основ языка Python и ознакомлением с некоторыми темами, рассматриваемыми далее.

Для этого я возьму довольно простое приложение, конструирующее базу данных, и проведу вас через различные этапы его создания: моделирование в интерактивном режиме, использование инструментов командной строки, создание интерфейса командной строки, создание графического интерфейса и создание простого веб-интерфейса. Попутно мы познакомимся с такими понятиями, как представление данных, сохранение объектов и объектно-ориентированное программирование (ООП); исследуем несколько альтернативных решений, к которым вернемся позднее; и рассмотрим некоторые основные идеи языка Python, которые вы должны знать, прежде чем продолжать чтение этой книги. В конечном итоге мы получим базу данных, хранящую экземпляры класса, которые можно будет просматривать и изменять с использованием различных интерфейсов.

Конечно, далее я затрону и дополнительные темы, но приемы, которые будут представлены здесь, применимы к некоторым прикладным областям, которые мы будем исследовать далее. Замечу также, что если что-то в программах из этой главы для вас останется непонятным, не волнуйтесь, так и должно быть – пока, по крайней мере. Здесь просто демонстрируются возможности Python. С недостающими подробностями вы познакомитесь достаточно скоро. А теперь начнем с небольшого развлечения.

Читатели четвертого издания книги «Изучаем Python» могут заметить в примере из этой главы знакомые черты – здесь участвуют те же персонажи, что и в главе про ООП в книге «Изучаем Python», а последние версии примера, основанные на использовании классов, по сути являются вариациями на ту же тему. Не боясь обвинений в избыточности, я здесь возвращаюсь к этому примеру по трем причинам: он вполне может использоваться для обзора основных возможностей языка; некоторые читатели этой книги не читали «Изучаем Python»; здесь этот пример получает дальнейшее развитие за счет добавления графического и веб-интерфейсов. Таким образом, эта глава начинается с того места, где закончилась книга «Изучаем Python», и помещает этот пример использования основных возможностей языка в контекст действующего приложения, что в общих чертах соответствует цели этой книги.

Постановка задачи

Представьте, что по некоторым причинам вам необходимо хранить информацию о людях. Это может быть адресная книга на вашем компьютере или, возможно, информация о служащих вашей фирмы. По каким-либо причинам вам требуется написать программу, которая сохраняла бы информацию об этих людях. Другими словами, вам требуется организовать сохранение записей в базе данных, чтобы обеспечить возможность долговременного хранения в компьютере списка людей с информацией о них.

Естественно, существует множество уже готовых программ для работы с базами данных. Однако, написав свою программу, вы получите полный контроль над тем, как она действует. Вы сможете добавлять в нее обработку специальных случаев, которые, возможно, не предусматриваются стандартным программным обеспечением. Вам не придется устанавливать базу данных и учиться пользоваться ею. И вам не придется ждать, пока поставщик программного обеспечения исправит ошибки или добавит новые особенности. Итак, вы решили написать на языке Python программу, управляющую информацией о людях.

Шаг 1: представление записей

Коль скоро мы собрались сохранять записи в базе данных, на самом первом этапе нам необходимо решить, как будут выглядеть эти записи. В языке Python имеется масса способов представления информации о людях. Зачастую для этих целей вполне достаточно бывает использовать объекты встроенных типов, такие как списки и словари, особенно если изначально не требуется предусматривать обработку сохраняемых данных.

Списки

Списки, например, позволяют сохранять информацию о людях упорядоченным способом. Запустите интерпретатор Python в интерактивном режиме и введите следующие две инструкции:

```
>>> bob = ['Bob Smith', 42, 30000, 'software']
>>> sue = ['Sue Jones', 45, 40000, 'hardware']
```

Мы только что создали две простые записи, представляющие информацию о Бобе (Bob) и Сью (Sue) (мои извинения, если вас действительно зовут Боб или Сью[1]). Каждая запись является списком с четырьмя элементами: имя, возраст, оклад и должность. Чтобы получить доступ к этим элементам, достаточно просто использовать операцию индексирования. Результат в примере ниже заключен в круглые скобки потому, что он является кортежем из двух результатов:

```
>>> bob[0], sue[2]            # получить имя и оклад
('Bob Smith', 40000)
```

В таком представлении записи легко обрабатывать – достаточно просто использовать операции над списками. Например, можно получить фамилию человека, разбив поле с именем по пробельному символу и отобрав последнюю часть, точно так же можно повысить оклад, изменив соответствующий список:

```
>>> bob[0].split()[-1]        # получить фамилию Боба
'Smith'
>>> sue[2] *= 1.25            # повысить оклад Сью на 25%
>>> sue
['Sue Jones', 45, 50000.0, 'hardware']
```

[1] Я вполне серьезно. На курсах по изучению Python, которые я веду уже довольно давно, я постоянно использовал имя «Bob Smith» (Боб Смит), возраст 40,5 лет и должности «developer» (разработчик) и «manager» (руководитель) для создания фиктивной записи в базе данных, пока на курсах в Чикаго я не встретил студента с именем Боб Смит (Bob Smith), которому было 40,5 лет, и который занимал должности разработчика и руководителя. В жизни иногда случаются курьезы.

Выражение, извлекающее фамилию в этом примере, обрабатывается интерпретатором слева направо: сначала извлекается строка с именем и фамилией Боба, затем строка разбивается по пробелам и преобразуется в список подстрок, а операция индексирования возвращает фамилию (разбейте это выражение на несколько операций, чтобы увидеть, как это происходит).

Первые замечания

Поскольку это первый пример программного кода в книге, необходимо сделать несколько практических замечаний:

- Этот программный код можно ввести в среде IDLE с графическим интерфейсом; после ввода команды `python` в командной строке (или той же команды с указанием полного пути к ней, если она не находится в системном списке путей поиска выполняемых файлов) и так далее.
- Символы >>> characters – это приглашение к вводу интерпретатора Python (эти символы не нужно вводить).
- Информационные строки, которые интерпретатор Python выводит при запуске, я опустил ради экономии места.
- Все примеры из этой книги я запускал под управлением Python 3.1; результаты работы примеров во всех версиях линейки 3.X должны быть одинаковыми (разумеется, исключая непредвиденные случаи внесения существенных изменений в Python).
- Большая часть примеров в этой книге, за исключением некоторых из них, демонстрирующих приемы системного программирования и интеграции с программным кодом на языке C, выполнялись в ОС Windows 7. Однако, благодаря переносимости Python, не имеет значения, в какой операционной системе будут опробоваться примеры, если иное не указано явно.

Если прежде вам не доводилось выполнять программный код на языке Python подобным способом, тогда обращайтесь за справочной информацией к вводным материалам, таким как книга «Изучаем Python». Далее в этой главе я сделаю несколько замечаний, касающихся запуска программного кода, хранящегося в файлах сценариев.

База данных в виде списка

К настоящему моменту мы всего создали всего лишь две переменных, но не базу данных. Чтобы объединить информацию о Бобе и Сью, мы могли бы просто включить ее в другой список:

```
>>> people = [bob, sue]        # ссылки в списке списков
>>> for person in people:
        print(person)
```

```
['Bob Smith', 42, 30000, 'software']
['Sue Jones', 45, 50000.0, 'hardware']
```

Теперь нашу базу данных представляет список `people`. Мы можем извлекать из него отдельные записи в соответствии с их позициями в списке и обрабатывать их в цикле:

```
>>> people[1][0]
'Sue Jones'

>>> for person in people:
        print(person[0].split()[-1])          # вывести фамилию
        person[2] *= 1.20                     # увеличить оклад на 20%

Smith
Jones

>>> for person in people: print(person[2])    # проверить новый размер оклада

36000.0
60000.0
```

Теперь, когда у нас имеется список, мы можем организовать выборку значений полей из записей с помощью более мощных инструментов выполнения итераций, присутствующих в языке Python, таких как генераторы списков, функция `map` и выражения-генераторы:

```
>>> pays = [person[2] for person in people]    # выбрать все оклады
>>> pays
[36000.0, 60000.0]

>>> pays = map((lambda x: x[2]), people)   # то же самое (в версии 3.X
>>> list(pays)                             # функция map возвращает генератор)
[36000.0, 60000.0]

>>> sum(person[2] for person in people)    # выражение-генератор,
96000.0                                    # sum - встроенная функция
```

Для добавления новых записей в базу данных вполне достаточно использовать обычные операции над списками, такие как `append` и `extend`:

```
>>> people.append(['Tom', 50, 0, None])
>>> len(people)
3
>>> people[-1][0]
'Tom'
```

Списки неплохо подходят для реализации нашей базы данных, и их возможностей вполне достаточно для некоторых программ, но они страдают рядом существенных недостатков. Во-первых, информация о Бобе и Сью в настоящий момент хранится в виде объектов в оператив-

ной памяти, и она будет утеряна сразу после завершения работы интерпретатора Python. Во-вторых, всякий раз, когда потребуется извлечь фамилию человека или повысить ему оклад, нам придется повторно вводить программный код, который мы только что видели. Это может вызвать определенные проблемы, если когда-нибудь поменяется алгоритм выполнения этих операций, – нам может потребоваться изменить наш программный код во многих местах. Мы вернемся к этим проблемам через несколько минут.

Обращение к полям по именам

Самый, пожалуй, существенный недостаток при использовании списков заключается в необходимости запоминать позиции полей: как иначе можно утверждать, что программный код, обращающийся к элементу записи с таинственным индексом 2, извлекает размер оклада? С точки зрения очевидности программного кода было бы лучше, если бы можно было присвоить каждому полю осмысленное имя.

Мы могли бы связать имена с позициями полей в записи, используя встроенную функцию `range`, которая генерирует набор последовательных целых чисел при использовании в контексте итераций (таких как операция присваивания последовательности ниже):

```
>>> NAME, AGE, PAY = range(3)          # 0, 1 и 2
>>> bob = ['Bob Smith', 42, 10000]
>>> bob[NAME]
'Bob Smith'
>>> PAY, bob[PAY]
(2, 10000)
```

Это решает проблему читаемости программного кода: три имени переменных, состоящих из заглавных символов, по сути, превратились в имена полей. Однако такое решение делает программный код зависящим от инструкции присваивания позиций именам полей, – мы должны помнить о необходимости обновлять ее при любом изменении структуры записи. Поскольку имена полей и записи никак не связаны между собой, они могут перестать соответствовать друг другу, и тогда возникнет необходимость вмешательства в программный код.

Кроме того, так как имена полей являются независимыми переменными, между записью в виде списка и именами полей отсутствует обратная связь. Имея одну только запись в виде списка, например, нельзя реализовать форматированный вывод значений полей с их именами. В случае с предыдущей записью без дополнительных ухищрений невозможно получить имя AGE из значения 42: вызов `bob.index(42)` вернет 1, значение переменной `AGE`, но не само имя AGE.

Можно было бы попробовать представлять записи в виде списков кортежей, где кортежи хранят не только значения полей, но их имена. Но

еще лучше было бы использовать списки списков, что позволило бы изменять поля (кортежи относятся к категории неизменяемых объектов). Ниже демонстрируется воплощение этой идеи на примере простых записей:

```
>>> bob = [['name', 'Bob Smith'], ['age', 42], ['pay', 10000]]
>>> sue = [['name', 'Sue Jones'], ['age', 45], ['pay', 20000]]
>>> people = [bob, sue]
```

Однако на самом деле этот прием не решает проблему, потому что для извлечения полей все равно необходимо использовать числовые индексы:

```
>>> for person in people:
        print(person[0][1], person[2][1])  # имя, оклад

Bob Smith 10000
Sue Jones 20000

>>> [person[0][1] for person in people]    # выборка имен
['Bob Smith', 'Sue Jones']

>>> for person in people:
        print(person[0][1].split()[-1])    # получить фамилию
        person[2][1] *= 1.10               # повысить оклад на 10%

Smith
Jones

>>> for person in people: print(person[2])

['pay', 11000.0]
['pay', 22000.0]
```

Все, чего мы добились, – добавили еще один уровень индексирования. Для достижения желаемого мы могли бы просматривать имена полей в цикле, отыскивая необходимые (в следующем цикле используется операция присваивания кортежа для распаковывания пар имя/значение):

```
>>> for person in people:
        for (name, value) in person:
            if name == 'name': print(value)    # поиск требуемого поля

Bob Smith
Sue Jones
```

Еще лучше было бы реализовать функцию, выполняющую всю работу за нас:

```
>>> def field(record, label):
        for (fname, fvalue) in record:
```

```
            if fname == label:                  # поиск поля по имени
                return fvalue

>>> field(bob, 'name')
'Bob Smith'
>>> field(sue, 'pay')
22000.0

>>> for rec in people:                          # вывести возраст всех людей
        print(field(rec, 'age'))                # в базе данных

42
45
```

Если двигаться дальше по этому пути, в конечном итоге можно получить набор функций, осуществляющих интерфейс к записям, отображающих имена полей в их значения. Однако если прежде вам приходилось программировать на языке Python, вы наверняка знаете, что существует более простой способ реализации такого рода ассоциаций, и вы уже наверняка догадались, в какую сторону мы направимся в следующем разделе.

Словари

Реализация записей на основе списков, как было показано в предыдущем разделе, вполне работоспособна, хотя и за счет некоторой потери производительности из-за необходимости выполнять поиск полей по именам (если вас волнуют потери, измеряемые миллисекундами). Однако если вы уже имеете некоторое знакомство с языком Python, вы должны знать, что существуют более эффективные и более удобные способы связывания имен полей с их значениями. Встроенные объекты словарей выглядят естественно:

```
>>> bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
>>> sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
```

Теперь `bob` и `sue` — это объекты, автоматически отображающие имена полей в их значения, и их использование делает программный код более простым и понятным. Нам не требуется запоминать, что означают числовые индексы, и мы даем интерпретатору возможность использовать его высокоэффективный алгоритм индексации словарей, чтобы отыскивать значения полей, ассоциированные с их именами:

```
>>> bob['name'], sue['pay']     # в отличие от bob[0], sue[2]
('Bob Smith', 40000)

>>> bob['name'].split()[-1]
'Smith'

>>> sue['pay'] *= 1.10
```

```
>>> sue['pay']
44000.0
```

Поскольку теперь при обращении к полям используются символические имена, программный код выглядит более осмысленным для всех, кто будет читать его (включая и вас).

Другие способы создания словарей

Словари являются настолько удобными объектами при программировании на языке Python, что было предусмотрено еще несколько способов их создания, отличающихся от традиционного синтаксиса литералов, продемонстрированного выше, – например, вызовом конструктора с именованными аргументами, при этом все ключи будут строками:

```
>>> bob = dict(name='Bob Smith', age=42, pay=30000, job='dev')
>>> sue = dict(name='Sue Jones', age=45, pay=40000, job='hdw')
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
>>> sue
{'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
```

заполнением словаря поле за полем (напомню, что для ключей словаря не предусматривается какой-то определенный порядок следования):

```
>>> sue = {}
>>> sue['name'] = 'Sue Jones'
>>> sue['age'] = 45
>>> sue['pay'] = 40000
>>> sue['job'] = 'hdw'
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
```

объединением двух списков, содержащих имена и значения:

```
>>> names = ['name', 'age', 'pay', 'job']
>>> values = ['Sue Jones', 45, 40000, 'hdw']
>>> list(zip(names, values))
[('name', 'Sue Jones'), ('age', 45), ('pay', 40000), ('job', 'hdw')]
>>> sue = dict(zip(names, values))
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
```

Словари можно даже создавать из последовательностей ключей и необязательного начального значения для всех ключей (этот способ удобно использовать для инициализации пустых словарей):

```
>>> fields = ('name', 'age', 'job', 'pay')
>>> record = dict.fromkeys(fields, '?')
>>> record
{'job': '?', 'pay': '?', 'age': '?', 'name': '?'}
```

Списки словарей

Независимо от способа создания словарей, нам все еще необходимо собрать словари-записи в базу данных. Здесь также можно использовать список, при условии, что нам не требуется обеспечить доступ по ключу на верхнем уровне:

```
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
>>> sue
{'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}

>>> people = [bob, sue]                                  # ссылки в списке
>>> for person in people:
        print(person['name'], person['pay'], sep=', ') # все имена, оклады

Bob Smith, 30000
Sue Jones, 40000

>>> for person in people:
        if person['name'] == 'Sue Jones':                # оклад Сью
            print(person['pay'])

40000
```

Здесь точно так же используются инструменты итераций, но вместо таинственных числовых индексов используются ключи (в терминах баз данных генератор списков и функция `map` в следующем примере возвращают проекцию базы данных по полю «name»):

```
>>> names = [person['name'] for person in people]  # выбирает имена
>>> names
['Bob Smith', 'Sue Jones']

>>> list(map((lambda x: x['name']), people))       # то же самое
['Bob Smith', 'Sue Jones']

>>> sum(person['pay'] for person in people)        # сумма всех окладов
70000
```

Интересно, что такие инструменты, как генераторы списков и выражения-генераторы, способны по своему удобству приблизиться к запросам в языке SQL, с тем отличием, что они манипулируют объектами в памяти:

```
>>> [rec['name'] for rec in people if rec['age'] >= 45]    # SQL-подобный
['Sue Jones']                                              # запрос

>>> [(rec['age'] ** 2 if rec['age'] >= 45 else rec['age']) for rec in people]
[42, 2025]
```

```
>>> G = (rec['name'] for rec in people if rec['age'] >= 45)
>>> next(G)
'Sue Jones'

>>> G = ((rec['age'] ** 2 if rec['age'] >= 45 else rec['age'])
                          for rec in people)
>>> G.__next__()
42
```

А так как словари являются обычными объектами, к этим записям можно также обращаться с использованием привычного синтаксиса:

```
>>> for person in people:
        print(person['name'].split()[-1])      # фамилия
        person['pay'] *= 1.10                  # повышение на 10%

Smith
Jones

>>> for person in people: print(person['pay'])

33000.0
44000.0
```

Вложенные структуры

В предыдущих примерах мы могли бы при желании избежать необходимости писать дополнительный программный код, извлекающий фамилии, еще больше структурировав наши записи. Поскольку в языке все объекты составных типов данных могут вкладываться друг в друга сколь угодно глубоко, мы легко можем конструировать чрезвычайно сложные структуры данных, используя простой синтаксис объектов, а Python сам позаботится о создании компонентов, связывании структур в памяти и освобождении памяти позднее. Это одно из значительных преимуществ таких языков сценариев, как Python.

В следующем примере демонстрируется более структурированная запись, содержащая вложенный словарь, список и кортеж внутри другого словаря:

```
>>> bob2 = {'name': {'first': 'Bob', 'last': 'Smith'},
            'age': 42,
            'job': ['software', 'writing'],
            'pay': (40000, 50000)}
```

Эта запись содержит вложенные структуры, поэтому для доступа к более низкому уровню мы просто будем использовать двойные индексы:

```
>>> bob2['name']                       # полное имя Боба
{'last': 'Smith', 'first': 'Bob'}
>>> bob2['name']['last']               # фамилия Боба
'Smith'
```

```
>>> bob2['pay'][1]                    # верхний предел оклада Боба
50000
```

Поле `name` здесь – это еще один словарь, поэтому вместо того чтобы разбивать строку для извлечения фамилии, мы просто используем операцию индексирования. Кроме того, сотрудники могут занимать несколько должностей, а также иметь верхний и нижний предел оклада. Фактически в подобных ситуациях Python превращается в своеобразный язык запросов – мы можем извлекать и изменять вложенные значения с применением обычных операций над объектами:

```
>>> for job in bob2['job']: print(job) # все должности, занимаемые Бобом
software
writing

>> bob2['job'][-1]                    # последняя должность Боба
'writing'
>>> bob2['job'].append('janitor')     # Боб получает новую должность
>>> bob2
{'job': ['software', 'writing', 'janitor'], 'pay': (40000, 50000), 'age': 42,
'name': {'last': 'Smith', 'first': 'Bob'}}
```

В расширении вложенного списка с помощью метода `append` нет ничего необычного, потому что в действительности он является независимым объектом. Такие вложенные конструкции могут пригодиться в более сложных приложениях. Однако, чтобы не усложнять примеры, мы сохраним прежнюю, плоскую структуру записей.

Словари словарей

И еще один поворот в реализации нашей базы данных с информацией о людях: мы можем расширить область применения словарей, задействовав еще один словарь для представления самой базы данных. То есть мы можем создать словарь словарей – внешний словарь будет играть роль базы данных, а вложенные словари – роль записей. В отличие от простого списка записей, база данных, представленная в виде словаря, позволит нам сохранять и извлекать записи с помощью символических ключей:

```
>>> bob = dict(name='Bob Smith', age=42, pay=30000, job='dev')
>>> sue = dict(name='Sue Jones', age=45, pay=40000, job='hdw')
>>> bob
{'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}

>>> db = {}
>>> db['bob'] = bob            # ссылки на словари в словаре
>>> db['sue'] = sue
>>>
>>> db['bob']['name']          # извлечь имя Боба
'Bob Smith'
```

```
>>> db['sue']['pay'] = 50000   # изменить оклад Сью
>>> db['sue']['pay']           # извлечь оклад Сью
50000
```

Обратите внимание, что такая организация позволяет нам обращаться к записям непосредственно, без необходимости выполнять поиск в цикле – мы получаем непосредственный доступ к записи с информацией о Бобе за счет использования ключа `bob`. Это действительно словарь словарей, хотя это и не заметно, если не вывести всю базу данных сразу (для подобных целей удобно использовать модуль `pprint` форматированного вывода):

```
>>> db
{'bob': {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}, 'sue':
{'pay': 50000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}}

>>> import pprint
>>> pprint.pprint(db)
{'bob': {'age': 42, 'job': 'dev', 'name': 'Bob Smith', 'pay': 30000},
'sue': {'age': 45, 'job': 'hdw', 'name': 'Sue Jones', 'pay': 50000}}
```

Если же возникнет необходимость последовательно обойти все записи в базе данных, можно воспользоваться итераторами словарей. В последних версиях Python реализован итератор словаря, который на каждой итерации в цикле `for` воспроизводит по одному ключу (для совместимости с более ранними версиями в циклах `for` можно также вместо простого имени `db` использовать явный вызов метода `db.keys`, но, так как в Python 3 метод `keys` возвращает генератор, конечный результат будет тот же самый):

```
>>> for key in db:
        print(key, '=>', db[key]['name'])

bob => Bob Smith
sue => Sue Jones

>>> for key in db:
        print(key, '=>', db[key]['pay'])

bob => 30000
sue => 50000
```

В процессе обхода доступ к отдельным записям можно получать с использованием операции индексирования по ключу:

```
>>> for key in db:
        print(db[key]['name'].split()[-1])
        db[key]['pay'] *= 1.10

Smith
Jones
```

или напрямую, организовав обход значений словаря:

```
>>> for record in db.values(): print(record['pay'])

33000.0
55000.0

>>> x = [db[key]['name'] for key in db]
>>> x
['Bob Smith', 'Sue Jones']

>>> x = [rec['name'] for rec in db.values()]
>>> x
['Bob Smith', 'Sue Jones']
```

А чтобы добавить новую запись, достаточно просто выполнить операцию присваивания по новому ключу. В конце концов – это всего лишь словарь:

```
>>> db['tom'] = dict(name='Tom', age=50, job=None, pay=0)
>>>
>>> db['tom']
{'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
>>> db['tom']['name']
'Tom'
>>> list(db.keys())
['bob', 'sue', 'tom']
>>> len(db)
3
>>> [rec['age'] for rec in db.values()]
[42, 45, 50]
>>> [rec['name'] for rec in db.values() if rec['age'] >= 45] # SQL-подобный
['Sue Jones', 'Tom']                                         # запрос
```

Наша база данных по-прежнему является объектом, хранящимся в оперативной памяти. Но, как оказывается, такой формат словаря словарей в точности соответствует формату, который используется системой сохранения объектов в файлах, – модулем `shelve` (с точки зрения грамматики английского языка этот модуль должен был бы называться *shelf*, но в Python термин *shelve*, обозначающий сохранение объектов, одновременно служит и названием соответствующего ему модуля). О том, как это делается, вы узнаете в следующем разделе.

Шаг 2: сохранение записей на длительное время

К настоящему моменту мы остановились на представлении нашей базы данных в виде словаря и попутно рассмотрели некоторые способы реализации структур данных в языке Python. Однако, как уже упоминалось выше, объекты, с которыми мы имели дело до сих пор, – временные объекты; они располагаются в оперативной памяти и исчезают бес-

следно после завершения работы интерпретатора Python или программы, создавшей их. Чтобы обеспечить долговременное хранение базы данных, ее необходимо сохранить в каком-нибудь файле.

Текстовые файлы

Один из способов обеспечить сохранность данных между запусками программы заключается в сохранении всех данных в простом файле, в виде отформатированного текста. Выработав определенную договоренность о формате представления данных, которая будет использоваться инструментами чтения и записи, мы сможем реализовать любую схему хранения.

Тестовый сценарий создания данных

Чтобы не продолжать дальнейшую работу в интерактивном режиме, напишем сценарий, инициализирующий базу данных, которую требуется сохранить (если вы уже имеете опыт работы с языком Python, то должны знать, что как только вы начинаете писать программный код, занимающий больше одной строки, работа в интерактивном режиме становится утомительной). Сценарий в примере 1.1 создает несколько записей и словарь базы данных, с которыми мы работали до сих пор, но так как он одновременно является модулем, мы сможем импортировать его и избавиться от необходимости всякий раз вводить программный код вручную. В некотором смысле этот модуль и является базой данных, но его реализация не поддерживает возможность изменения данных автоматически или конечным пользователем.

Пример 1.1. PP4E\Preview\initdata.py

```
# инициализировать данные для последующего сохранения в файлах

# записи
bob = {'name': 'Bob Smith', 'age': 42, 'pay': 30000, 'job': 'dev'}
sue = {'name': 'Sue Jones', 'age': 45, 'pay': 40000, 'job': 'hdw'}
tom = {'name': 'Tom',       'age': 50, 'pay': 0,     'job': None}

# база данных
db = {}
db['bob'] = bob
db['sue'] = sue
db['tom'] = tom

if __name__ == '__main__':     # если запускается, как сценарий
    for key in db:
        print(key, '=>\n ', db[key])
```

Как обычно, проверка переменной `__name__` в конце примера 1.1 возвращает `true`, только если файл был запущен как самостоятельный сценарий, а не был импортирован как модуль. Если запустить пример как

самостоятельный сценарий (например, из командной строки, щелчком на ярлыке или из среды IDLE), будет выполнен программный код теста в теле условной инструкции, который выведет содержимое базы данных в поток стандартного вывода (напомню, что функция `print` использует этот поток по умолчанию).

Ниже приводится пример запуска сценария из командной строки в ОС Windows. В окне Командная строка (Command Prompt) выполните команду `cd`, чтобы перейти в каталог со сценарием. На других платформах используйте аналогичную программу-консоль:

```
...\PP4E\Preview> python initdata.py
bob =>
   {'job': 'dev', 'pay': 30000, 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'job': 'hdw', 'pay': 40000, 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'job': None, 'pay': 0, 'age': 50, 'name': 'Tom'}
```

Соглашения об именовании файлов

Это наш первый файл (он же «сценарий») с исходными текстами, поэтому здесь необходимо сделать три замечания по использованию примеров из книги:

- Текст `...\PP4E\Preview>` в первой строке предыдущего примера обозначает приглашение к вводу в командной строке и может отличаться в разных платформах. Вам необходимо ввести лишь текст, следующий за этим приглашением (`python initdata.py`).
- Во всех примерах в этой книге текст системной подсказки к вводу содержит путь к каталогу в загружаемом пакете с примерами, внутри которого должна запускаться указанная команда. При запуске сценария из командной строки убедитесь, что текущим рабочим каталогом является каталог *PP4E\Preview*. Это может иметь значение для примеров, использующих файлы в рабочем каталоге.
- Кроме того, подписи, предшествующие примерам с листингами программного кода из файлов, сообщают, где находится файл в пакете с примерами. Так, подпись к примеру 1.1 выше сообщает, что полное имя сценария в дереве каталогов имеет вид *PP4E\Preview\initdata.py*.

Мы будем пользоваться этими соглашениями на протяжении всей книги – в предисловии описано, как получить примеры, если вы собираетесь работать с ними. Иногда, особенно в части книги о системном программировании, я буду указывать в приглашении к вводу более полный путь к каталогу, если это будет необходимо, чтобы уточнить контекст выполнения (например, префикс «C:\» в Windows или дополнительные имена каталогов).

Замечания по поводу сценариев

Выше я уже давал несколько замечаний по поводу интерактивного режима. Теперь, когда мы начинаем использовать файлы сценариев, я также хочу дать несколько общих замечаний по использованию сценариев Python:

- На некоторых платформах может потребоваться вводить полный путь к каталогу с программой на языке Python. Если путь к выполняемому файлу интерпретатора Python отсутствует в системном пути поиска, замените в Windows, например, команду `python` на `C:\Python31\python` (здесь предполагается, что вы пользуетесь версией Python 3.1).
- В большинстве систем Windows вообще не обязательно вводить команду `python` – чтобы запустить сценарий, вполне достаточно ввести только имя файла, поскольку интерпретатор Python обычно регистрируется, как программа для открытия файлов с расширением «.py».
- Кроме того, файлы сценариев можно запускать в стандартной среде IDLE (откройте файл и запустите его, воспользовавшись меню `Run` (Запустить) в окне редактирования файла) или похожим способом в любой другой среде разработки программ на языке Python IDE (например, в Komodo, Eclipse, NetBeans или Wing IDE).
- Если вы собираетесь запускать файл в Windows щелчком мыши на ярлыке, не забудьте добавить вызов функции `input()` в конец сценария, чтобы окно с выводом программы не закрылось после ее завершения. В других системах, чтобы обеспечить возможность запуска сценария щелчком на ярлыке, может потребоваться добавить в его начало строку `#!` и сделать файл выполняемым с помощью команды `chmod`.

Далее я буду исходить из предположения, что вы в состоянии запустить программный код на языке Python каким-либо способом. Однако если вы столкнетесь с трудностями, за полной информацией о способах запуска программ на языке Python обращайтесь к другим книгам, таким как «Изучаем Python».

Сценарий записи/чтения данных

Теперь осталось лишь реализовать сохранение всех данных из памяти в файле. Добиться этого можно несколькими способами. Самый простой из них заключается в том, чтобы сохранять записи по одной и вставлять между ними разделители, которые можно будет использовать при загрузке данных, чтобы отделять записи друг от друга. В примере 1.2 показан один из способов воплощения этой идеи.

Пример 1.2. PP4E\Preview\make_db_file.py

```
"""
Сохраняет в файл базу данных, находящуюся в оперативной памяти, используя
собственный формат записи; предполагается, что в данных отсутствуют строки
'endrec.', 'enddb.' и '=>'; предполагается, что база данных является словарем
словарей; внимание: применение функции eval может быть опасным – она
выполняет строки как программный код; с помощью функции eval() можно также
реализовать сохранение словарей-записей целиком; кроме того, вместо вызова
print(key,file=dbfile) можно использовать вызов dbfile.write(key + '\n');
"""

dbfilename = 'people-file'
ENDDB = 'enddb.'
ENDREC = 'endrec.'
RECSEP = '=>'

def storeDbase(db, dbfilename=dbfilename):
    "сохраняет базу данных в файл"
    dbfile = open(dbfilename, 'w')
    for key in db:
        print(key, file=dbfile)
        for (name, value) in db[key].items():
            print(name + RECSEP + repr(value), file=dbfile)
        print(ENDREC, file=dbfile)
    print(ENDDB, file=dbfile)
    dbfile.close()

def loadDbase(dbfilename=dbfilename):
    "восстанавливает данные, реконструируя базу данных"
    dbfile = open(dbfilename)
    import sys
    sys.stdin = dbfile
    db = {}
    key = input()
    while key != ENDDB:
        rec = {}
        field = input()
        while field != ENDREC:
            name, value = field.split(RECSEP)
            rec[name] = eval(value)
            field = input()
        db[key] = rec
        key = input()
    return db

if __name__ == '__main__':
    from initdata import db
    storeDbase(db)
```

Это достаточно сложная программа, отчасти потому, что в ней реализованы обе операции, сохранения и загрузки, а отчасти потому, что эти операции реализованы не самым простым способом. Как будет показано ниже, существуют более простые способы сохранения объектов в файл и чтения их из файла по сравнению с форматированием и парсингом данных вручную.

Однако для реализации простых задач такой подход вполне оправдан. При запуске примера 1.2 как сценария база данных будет сохранена в файл. Он ничего не выводит на экран, но мы можем проверить содержимое файла базы данных в интерактивной оболочке после выполнения сценария внутри IDLE или в окне консоли (так как файл базы данных появится в текущем рабочем каталоге):

```
...\PP4E\Preview> python make_db_file.py
...\PP4E\Preview> python
>>> for line in open('people-file'):
...     print(line, end='')
...
bob
job=>'dev'
pay=>30000
age=>42
name=>'Bob Smith'
endrec.
sue
job=>'hdw'
pay=>40000
age=>45
name=>'Sue Jones'
endrec.
tom
job=>None
pay=>0
age=>50
name=>'Tom'
endrec.
enddb.
```

Этот файл хранит содержимое базы данных с дополнительными элементами форматирования. Сами данные берутся из тестовой базы данных, созданной модулем, представленным в примере 1.1, который импортируется программным кодом самопроверки в примере 1.2. С точки зрения практического применения, пример 1.2 сам мог бы импортироваться и использоваться для сохранения различных баз данных.

Обратите внимание, что форматирование сохраняемых данных выполняется с помощью функции `repr`, а обратное преобразование прочитанных данных – с помощью функции `eval`, которая интерпретирует

входную строку как программный код на языке Python. Это позволяет сохранять и воссоздавать такие виды данных, как объект None, но этот способ небезопасен. Не следует использовать функцию eval, если нет уверенности, что база данных не содержит злонамеренный программный код. Однако в нашем случае нет причин для волнений.

Вспомогательные сценарии

Ниже приводятся дополнительные сценарии, которые можно использовать для тестирования. Сценарий в примере 1.3 выполняет загрузку базы данных из файла.

Пример 1.3. PP4E\Preview\dump_db_file.py

```
from make_db_file import loadDbase
db = loadDbase()
for key in db:
    print(key, '=>\n ', db[key])
print(db['sue']['name'])
```

А сценарий в примере 1.4 загружает базу данных, вносит в нее изменения и сохраняет ее обратно в файл.

Пример 1.4. PP4E\Preview\update_db_file.py

```
from make_db_file import loadDbase, storeDbase
db = loadDbase()
db['sue']['pay'] *= 1.10
db['tom']['name'] = 'Tom Tom'
storeDbase(db)
```

Ниже приводится пример запуска сценариев *dump_db_file.py* и *update_db_file.py* из командной строки, где видно, что между запусками сценария *dump_db_file.py* изменяются оклад Сью и имя Тома. Обратите внимание, что после завершения каждого из сценариев данные сохраняются, – это обусловлено тем, что наши объекты просто загружаются и сохраняются в текстовом файле:

```
...\PP4E\Preview> python dump_db_file.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones

...\PP4E\Preview> python update_db_file.py
...\PP4E\Preview> python dump_db_file.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
```

```
sue =>
   {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom Tom'}
Sue Jones
```

В настоящий момент, чтобы внести в базу данных какие-либо изменения, необходимо написать отдельный программный код, в виде сценария или в интерактивной оболочке (далее в этой главе будут представлены более удачные решения с универсальным интерфейсом командной строки, графическим интерфейсом и веб-интерфейсом). Но, как бы то ни было, наш текстовый файл является базой данных, содержащей записи. Однако, как мы увидим в следующем разделе, мы только что проделали массу ненужной работы.

Модуль pickle

Реализация базы данных на основе текстового файла, представленная в предыдущем разделе, вполне работоспособна, но она имеет ряд существенных ограничений. Во-первых, чтобы получить доступ всего к одной записи, необходимо загрузить всю базу данных из файла, а после некоторых изменений обратно в файл необходимо записывать всю базу данных. Данное ограничение можно обойти, если каждую запись сохранять в отдельном файле, но это еще больше осложнит реализацию программы.

Во-вторых, решение на основе текстового файла предполагает, что символы, выполняющие роль разделителей записей, не должны появляться в самих данных: если, к примеру, данные могут содержать последовательность символов =>, предложенное решение окажется непригодным. Мы могли бы обойти это ограничение, сохраняя записи в формате XML, а для загрузки данных используя инструменты для работы с форматом XML, входящие в состав Python, с которыми мы познакомимся далее в этой книге. Использование тегов XML позволило бы избежать конфликтов с фактическими данными в текстовом виде, но необходимость создания и парсинга XML снова привела бы к усложнению программы.

Хуже всего, пожалуй, то, что решение на основе текстового файла оказывается слишком сложным, не будучи при этом достаточно универсальным: оно привязано к организации базы данных в виде словаря словарей и не способно работать с базами данных, имеющими другую структуру, без существенного расширения реализации. Было бы здорово, если бы существовал универсальный инструмент, способный преобразовывать любые типы данных Python в формат, который можно было бы сохранять в файл за один шаг.

Именно для этого разработан модуль `pickle`. Этот модуль преобразует объект Python, находящийся в оперативной памяти, в последователь-

ность или в строку байтов, которую можно записать в любой объект, подобный файлу. Кроме того, модуль pickle знает, как восстановить оригинальный объект в памяти, получив последовательность байтов, то есть мы получаем обратно тот же самый объект. В некотором смысле модуль pickle позволяет избежать необходимости разрабатывать специальные форматы представления данных – последовательный формат, реализованный в этом модуле, достаточно универсален и эффективен для большинства применений. При использовании модуля pickle отпадает необходимость вручную преобразовывать объекты перед сохранением и анализировать сложный формат представления данных, чтобы получить исходные объекты. Прием, основанный на использовании модуля pickle, напоминает прием, основанный на использовании формата XML, но он является не только более характерным для Python, но и более простым в реализации.

Другими словами, применение модуля pickle позволит нам сохранять и извлекать объекты Python за один шаг, благодаря чему мы сможем обрабатывать записи, используя привычный синтаксис языка Python. Невзирая на сложность реализации, модуль pickle удивительно прост в использовании. В примере 1.5 демонстрируется, как с помощью этого модуля можно сохранять записи в файле.

Пример 1.5. PP4E\Preview\make_db_pickle.py

```
from initdata import db
import pickle
dbfile = open('people-pickle', 'wb') # в версии 3.X следует использовать
pickle.dump(db, dbfile)              # двоичный режим работы с файлами, так как
dbfile.close()                       # данные имеют тип bytes, а не str
```

Если запустить этот сценарий, он сохранит всю базу данных (словарь словарей, который создается сценарием из примера 1.1) в файл с именем *people-pickle* в текущем рабочем каталоге. В процессе работы модуль pickle преобразовывает объект в строку. В примере 1.6 демонстрируется, как можно реализовать доступ к сохраненной базе данных после ее создания, – достаточно просто открыть файл и передать его модулю pickle, который восстановит объект из последовательного представления.

Пример 1.6. PP4E\Preview\dump_db_pickle.py

```
import pickle
dbfile = open('people-pickle', 'rb') # в версии 3.X следует использовать
db = pickle.load(dbfile)             # двоичный режим работы с файлами
for key in db:
    print(key, '=>\n ', db[key])
print(db['sue']['name'])
```

Ниже приводится пример запуска этих двух сценариев из командной строки. Естественно, эти сценарии можно запустить и в среде IDLE, чтобы в интерактивном сеансе открыть и исследовать файл, созданный модулем pickle:

```
...\PP4E\Preview> python make_db_pickle.py
...\PP4E\Preview> python dump_db_pickle.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

Внесение изменений в базу данных, сохраненную с помощью модуля `pickle`, выполняется точно так же, как и при использовании текстового файла, созданного вручную, за исключением того, что все необходимые преобразования выполняются стандартным модулем. Как это делается, демонстрирует пример 1.7.

Пример 1.7. PP4E\Preview\update-db-pickle.py

```
import pickle
dbfile = open('people-pickle', 'rb')
db = pickle.load(dbfile)
dbfile.close()

db['sue']['pay'] *= 1.10
db['tom']['name'] = 'Tom Tom'

dbfile = open('people-pickle', 'wb')
pickle.dump(db, dbfile)
dbfile.close()
```

Обратите внимание, что после изменения записи в файл сохраняется вся база данных целиком, как и при использовании простого текстового файла; это может занимать продолжительное время, если база данных имеет значительный объем, но мы пока не будем беспокоиться об этом. Ниже приводится пример запуска сценариев *dump_db_pickle.py* и *update_db_pickle.py* – как и в предыдущем разделе, измененный оклад Сью и имя Тома сохраняются между вызовами сценариев, потому что записываются обратно в файл (но на этот раз с помощью модуля `pickle`):

```
...\PP4E\Preview> python update_db_pickle.py
...\PP4E\Preview> python dump_db_pickle.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom Tom'}
Sue Jones
```

Как мы узнаем в главе 17, модуль `pickle` поддерживает объекты практически любых типов – списки, словари, экземпляры классов, вложен-

ные структуры и многие другие. Там же мы узнаем о текстовых и двоичных протоколах преобразования сохраняемых данных. В Python 3 для представления сохраненных данных все протоколы используют объекты типа `bytes`, чем обусловлена необходимость открывать файлы `pickle` в двоичном режиме, независимо от используемого протокола. Кроме того, как будет показано далее в этой главе, модуль `pickle` и его формат представления данных используется модулем `shelve` и базами данных ZODB, а в случае экземпляров классов сохраняются не только данные в объектах, но и их поведение.

Модуль `pickle` фактически является гораздо более универсальным, чем можно было бы заключить из представленных примеров. Поскольку сериализованные данные принимаются любыми объектами, поддерживающими интерфейс, совместимый с файлами, методы `dump` и `load` модуля `pickle` могут использоваться для передачи объектов Python через различные среды распространения информации. С помощью сетевых сокетов, например, можно организовать передачу сериализованных объектов Python по сети и тем самым обеспечить альтернативу более тяжелым протоколам, таким как SOAP и XML-RPC.

Работа модуля pickle с отдельными записями

Как упоминалось выше, один из потенциальных недостатков примеров, представленных в этом разделе до настоящего момента, состоит в том, что они могут оказаться слишком медленными при работе с очень большими базами данных: так как для изменения единственной записи необходимо загружать и сохранять базу данных целиком, при таком решении значительная часть времени будет тратиться впустую. Мы могли бы избежать этого, предусмотрев сохранение каждой записи базы данных в отдельном файле. Следующие три примера демонстрируют, как это можно реализовать, – сценарий из примера 1.8 сохраняет каждую запись в отдельном файле, где в качестве имени файла используется уникальный ключ записи, к которому добавляется расширение *.pkl* (он создает файлы *bob.pkl*, *sue.pkl* и *tom.pkl* в текущем рабочем каталоге).

Пример 1.8. PP4E\Preview\make_db_pickle_recs.py

```
from initdata import bob, sue, tom
import pickle
for (key, record) in [('bob', bob), ('tom', tom), ('sue', sue)]:
    recfile = open(key + '.pkl', 'wb')
    pickle.dump(record, recfile)
    recfile.close()
```

Следующий сценарий, представленный в примере 1.9, выводит содержимое всей базы данных, используя модуль `glob` для подстановки имен файлов и тем самым для выбора всех файлов с расширением *.pkl*, присутствующих в текущем каталоге. Чтобы загрузить единственную запись, мы открываем файл этой записи и выполняем преобразование

содержимого файла с помощью модуля pickle. Теперь, чтобы получить одну запись, нам необходимо загрузить файл единственной записи, а не всю базу данных.

Пример 1.9. PP4E\Preview\dump_db_pickle_recs.py

```
import pickle, glob
for filename in glob.glob('*.pkl'):    # для 'bob','sue','tom'
    recfile = open(filename, 'rb')
    record = pickle.load(recfile)
    print(filename, '=>\n ', record)

suefile = open('sue.pkl', 'rb')
print(pickle.load(suefile)['name'])    # извлечь имя Сью
```

Наконец, сценарий в примере 1.10 обновляет содержимое базы данных, извлекая запись из ее файла, изменяя объект в памяти и затем сохраняя его обратно в файл с помощью модуля pickle. На этот раз для внесения изменений необходимо извлечь и сохранить единственную запись, а не всю базу данных.

Пример 1.10. PP4E\Preview\update_db_pickle_recs.py

```
import pickle
suefile = open('sue.pkl', 'rb')
sue = pickle.load(suefile)
suefile.close()
sue['pay'] *= 1.10
suefile = open('sue.pkl', 'wb')
pickle.dump(sue, suefile)
suefile.close()
```

Ниже приводится пример запуска сценариев, реализующих решение, когда каждая запись сохраняется в отдельном файле. Результаты получаются те же, что и прежде, но теперь уникальные ключи в базе данных играют роль имен файлов. При такой реализации файловая система превращается в своеобразный словарь верхнего уровня – имена файлов обеспечивают прямой доступ к отдельным записям.

```
...\PP4E\Preview> python make_db_pickle_recs.py
...\PP4E\Preview> python dump_db_pickle_recs.py
bob.pkl =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue.pkl =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom.pkl =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones

...\PP4E\Preview> python update_db_pickle_recs.py
...\PP4E\Preview> python dump_db_pickle_recs.py
```

```
bob.pkl =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue.pkl =>
   {'pay': 44000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom.pkl =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

Модуль shelve

Сохранение объектов в файлах, как было показано в предыдущем разделе, является оптимальным решением для многих приложений. Фактически некоторые приложения используют модуль `pickle` для передачи объектов Python через сетевые сокеты, как более простую альтернативу сетевым протоколам веб-служб, таким как SOAP и XML-RPC (они также поддерживаются в Python, но являются более тяжеловесными по сравнению с модулем `pickle`).

Кроме того, если допустить, что файловая система способна хранить необходимое нам количество файлов, реализация, сохраняющая каждую запись в отдельном файле, устраняет необходимость загрузки и сохранения всей базы данных при каждом изменении. Однако, когда действительно необходимо иметь возможность доступа к записям по ключу, можно использовать более высокоуровневый инструмент: модуль `shelve`.

Модуль `shelve` автоматически сохраняет и загружает объекты из хранилища, обеспечивающего доступ по ключу. Хранилища напоминают словари; их необходимо открывать, и они автоматически сохраняются после завершения программы. Поскольку хранилища обеспечивают доступ к хранимым записям по ключу, отпадает необходимость создавать отдельные файлы для каждой записи – модуль `shelve` автоматически разделяет записи и извлекает и обновляет только те записи, к которым осуществляется доступ или которые изменяются. Таким образом модуль `shelve` обеспечивает решение, напоминающее решение, сохраняющее каждую запись в отдельном файле, но более простое в использовании.

Интерфейс модуля `shelve` так же прост, как и интерфейс модуля `pickle`: хранилища, создаваемые модулем `shelve`, идентичны словарям с дополнительными методами `open` и `close`. В программном коде объекты хранилищ действительно выглядят, как словари, содержимое которых сохраняется после завершения программы. А все операции по отображению содержимого хранилища в файл и из файла выполняются интерпретатором Python. Например, сценарий в примере 1.11 демонстрирует, как можно сохранить объекты из словаря в хранилище, созданном с помощью модуля `shelve`.

Пример 1.11. PP4E\Preview\make_db_shelve.py

```
from initdata import bob, sue
import shelve
db = shelve.open('people-shelve')
db['bob'] = bob
db['sue'] = sue
db.close()
```

Этот сценарий создаст в текущем каталоге один или более файлов, имена которых начинаются с префикса *people-shelve* (в ОС Windows, под управлением Python 3.1, сценарий создаст файлы *people-shelve.bak*, *people-shelve.dat* и *people-shelve.dir*). Вы не должны удалять эти файлы (они составляют вашу базу данных!), а чтобы получить доступ к этому хранилищу в других сценариях, необходимо использовать то же самое имя базы. Сценарий в примере 1.12, например, повторно открывает хранилище и последовательно извлекает хранящиеся в нем записи.

Пример 1.12. PP4E\Preview\dump_db_shelve.py

```
import shelve
db = shelve.open('people-shelve')
for key in db:
    print(key, '=>\n ', db[key])
print(db['sue']['name'])
db.close()
```

Мы по-прежнему имеем словарь словарей, но на этот раз словарем верхнего уровня является хранилище, отображаемое в файл. Всякий раз, когда происходит обращение к элементу в хранилище, модуль `shelve` выполняет необходимые операции с файловой системой, обеспечивающей доступ по ключу, и использует модуль `pickle` для сериализации и десериализации объектов. Однако, с точки зрения программиста, хранилище – это всего лишь словарь, обладающий возможностью сохраняться между вызовами программы. Сценарий в примере 1.13 демонстрирует, как можно реализовать изменение данных в хранилище.

Пример 1.13. PP4E\Preview\update_db_shelve.py

```
from initdata import tom
import shelve
db = shelve.open('people-shelve')
sue = db['sue']                 # извлекает объект sue
sue['pay'] *= 1.50
db['sue'] = sue                 # изменяет объект sue
db['tom'] = tom                 # добавляет новую запись
db.close()
```

Обратите внимание, что в этом примере сначала по ключу извлекается объект `sue`, затем он изменяется в памяти и снова сохраняется в хранилище по ключу. Так действуют хранилища по умолчанию, однако более

совершенные системы хранения, такие как ZODB, о которой рассказывается в главе 17, могут действовать иначе. Как мы узнаем позднее, метод `shelve.open` в подобных системах имеет дополнительный именованный аргумент `writeback`. Если в этом аргументе передать значение `True`, все загруженные записи будут сохраняться в кэше и автоматически записываться обратно в файл при закрытии хранилища. Благодаря этому не требуется вручную записывать изменения обратно в хранилище, но при этом увеличивается потребление памяти, а сама операция закрытия может занимать продолжительное время.

Обратите также внимание на необходимость явного закрытия хранилища. Нам не требуется указывать флаги режимов в вызове метода `shelve.open` (по умолчанию он создает новое хранилище, если это необходимо, и открывает существующее хранилище для чтения и записи), однако некоторые механизмы, обеспечивающие доступ к содержимому файлов по ключу, требуют вызова метода `close`, чтобы сбросить на диск выходные буферы с изменениями.

Наконец, ниже приводится пример запуска сценариев, опирающихся на использование модуля `shelve`, которые создают, изменяют и извлекают записи. Записи по-прежнему реализованы как словари, но сама база данных теперь является хранилищем, напоминающим словарь, который автоматически сохраняет свое содержимое в файле:

```
...\PP4E\Preview> python make_db_shelve.py
...\PP4E\Preview> python dump_db_shelve.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 40000, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
Sue Jones

...\PP4E\Preview> python update_db_shelve.py
...\PP4E\Preview> python dump_db_shelve.py
bob =>
   {'pay': 30000, 'job': 'dev', 'age': 42, 'name': 'Bob Smith'}
sue =>
   {'pay': 60000.0, 'job': 'hdw', 'age': 45, 'name': 'Sue Jones'}
tom =>
   {'pay': 0, 'job': None, 'age': 50, 'name': 'Tom'}
Sue Jones
```

После выполнения сценариев *update_db_shelve.py* и *dump_db_shelve.py* можно заметить, что была добавлена новая запись с ключом `tom` и на 50 процентов был увеличен оклад Сью. Эти изменения сохраняются между запусками сценариев, потому что записи-словари отображаются модулем `shelve` во внешний файл хранилища. (Этот сценарий особенно хорош для Сью – у нее могло бы появиться желание почаще запускать этот сценарий с помощью планировщика cron в Unix или поместив его в папку Автозагрузка (Startup) с помощью `msconfig` в Windows...)

Что в имени тебе моем?

Удивительно, но часто остается тайной, что свое название язык Python получил благодаря британскому телевизионному комедийному сериалу «Monty Python's Flying Circus», появившемуся на экранах в 1970-х годах. Фольклор сообщества Python утверждает, что Гвидо ван Россум (Guido van Rossum), создатель Python, смотрел повторные показы этого сериала как раз в то время, когда подбирал название для нового языка программирования, который он разрабатывал. И, как говорят в шоу-бизнесе: «остальное уже история».

Такая наследственность часто является причиной появления в примерах и обсуждениях ссылок на комедийную игру. Например, в сценариях часто используется имя Brian; словами *spam* (консервированный фарш), *lumberjack* (лесоруб) и *shrubbery* (кустарник), получившими специальное значение, называют пользователей Python; а презентации иногда называют «испанской инквизицией». Как правило, когда пользователь Python начинает произносить фразы, не имеющие отношения к реальности, они оказываются заимствованными из сериала или фильмов с участием персонажа Monty Python. Некоторые из этих фраз могут встретиться даже в этой книге. Конечно, чтобы писать программы на языке Python, необязательно бежать и брать в прокате «The Meaning of Life» или «The Holy Grail», но и хуже от этого не будет.

Имя «Python» быстро прижилось, тем не менее его заимствование стало причиной интересных курьезов. Например, когда в 1994 году возникла телеконференция по Python, comp.lang.python, первые несколько недель она практически полностью была оккупирована желающими обсуждать темы, касающиеся телевизионной постановки. Позднее специальное приложение к журналу «Linux Journal», касающееся Python, стало сопровождаться фотографией Гвидо, облаченного в обязательную «красную форму».

В списке рассылки Python все еще появляются случайные письма от поклонников сериала. Например, в одном письме невинно предлагалось обмениваться сценариями с участием персонажа Monty Python. Если бы автор письма понимал назначение форума, то хотя бы указал, могут ли они выполняться в разных операционных системах или нет.

Шаг 3: переход к ООП

Давайте отвлечемся на минутку и посмотрим, куда мы пришли. Итак, в настоящий момент у нас имеется две реализации базы данных: на

основе модуля shelve и на основе модуля pickle, в которой каждая запись сохраняется в отдельном файле, – этого вполне достаточно для хранения простых данных. Наши записи представлены простыми словарями, которые обеспечивают более простой и понятный доступ к полям, чем списки (то есть не по числовым индексам, а по ключам). Однако словари имеют некоторые ограничения, которые могут оказаться существенными по мере разработки программы.

Во-первых, у нас не предусмотрено место для централизованного хранения логики обработки записей. Операции извлечения фамилии и увеличения оклада, например, могут выполняться так:

```
>>> import shelve
>>> db = shelve.open('people-shelve')
>>> bob = db['bob']
>>> bob['name'].split()[-1]    # вернет фамилию Боба
'Smith'
>>> sue = db['sue']
>>> sue['pay'] *= 1.25         # увеличит оклад Сью
>>> sue['pay']
75000.0
>>> db['sue'] = sue
>>> db.close()
```

Такое решение вполне пригодно для небольших программ. Но если когда-нибудь потребуется изменить логику извлечения фамилии или увеличения оклада, нам, возможно, придется обновить подобный программный код во множестве мест в программе. На практике даже просто отыскать все такие фрагменты может оказаться достаточно сложным делом – копирование одинаковых фрагментов программного кода рано или поздно обязательно даст знать о себе.

Подобные фрагменты предпочтительнее скрывать, то есть *инкапсулировать*. Эти операции можно было бы реализовать в виде функций в одном модуле и тем самым избежать избыточности, но при таком подходе функции никак не будут связаны с самими записями. Нам же желательно связать логику обработки с данными, хранящимися в базе данных, чтобы ее проще было понимать, отлаживать и многократно использовать.

Другой недостаток использования словарей для представления записей заключается в том, что со временем их становится трудно расширять. Например, представьте, что имеется набор полей данных или различные процедуры, увеличивающие оклад для разных сотрудников поразному (например, кто-то может получать ежегодную прибавку, а кто-то – нет). Если нам когда-нибудь потребуется расширить программу, сделать это будет очень сложно, так как нет простого и естественного способа расширить простые словари. С учетом дальнейшего роста нам

хотелось бы, чтобы наше программное обеспечение предусматривало возможность расширения и настройки естественными способами.

Если вы уже погружались в изучение Python, то, наверное, знаете, что это тот случай, когда начинает проявляться привлекательность ООП:

Структурирование

Благодаря ООП появляется возможность связать логику обработки записей с самими записями – классы представляют собой программные единицы, объединяющие логику и данные, а наследование позволяет легко избежать избыточности программного кода.

Инкапсуляция

Благодаря ООП можно скрыть детали реализации таких операций, как обработка имени или увеличение оклада, внутри методов – то есть в дальнейшем мы легко сможем изменять реализацию методов, не влияя на работоспособность программного кода, использующего их.

Специализация

Применение ООП обеспечивает естественный способ дальнейшего расширения. Классы могут расширяться и специализироваться за счет создания новых подклассов, без изменения или нарушения работоспособности существующего программного кода.

Таким образом, в объектно-ориентированном программировании мы специализируем и повторно используем программный код, а не переписываем его заново. ООП в Python является дополнительной возможностью, которая, честно признаться, лучше подходит для решения стратегических, а не тактических задач. ООП лучше подходит, когда у вас имеется время для предварительного планирования, что может оказаться непозволительной роскошью, когда ваши пользователи уже начали штурмовать ворота.

Преимущества структурирования и повторного использования программного кода в крупных системах, которые продолжают развиваться в течение длительного времени, перевешивают затраты на изучение ООП и способны существенно сократить время разработки. Даже в нашем простом случае возможность специализации и снижения избыточности, которую дают классы, может оказаться решающим преимуществом.

Использование классов

ООП в Python отличается простотой использования, в значительной степени благодаря динамической модели типов. Фактически программировать в объектно-ориентированном стиле настолько просто, что я сразу же перейду к примеру: пример 1.14 реализует наши записи уже не в виде словарей, а в виде экземпляров класса.

Пример 1.14. PP4E\Preview\person_start.py

```
class Person:
    def __init__(self, name, age, pay=0, job=None):
        self.name = name
        self.age  = age
        self.pay  = pay
        self.job  = job

if __name__ == '__main__':
    bob = Person('Bob Smith', 42, 30000, 'software')
    sue = Person('Sue Jones', 45, 40000, 'hardware')
    print(bob.name, sue.pay)

    print(bob.name.split()[-1])
    sue.pay *= 1.10
    print(sue.pay)
```

Это очень простой класс – он содержит единственный метод-конструктор, заполняющий экземпляр класса данными, переданными в виде аргументов при обращении к имени класса. Тем не менее этого вполне достаточно для представления записи, а кроме того, сюда уже можно добавить такие элементы, как значения по умолчанию для полей `pay` и `job`, чего нельзя сделать в словарях. Программный код самотестирования в конце этого файла создает два экземпляра класса (две записи) и обращается к их атрибутам (полям). Ниже приводится вывод, полученный в результате запуска этого сценария в среде IDLE (при запуске из командной строки результаты получаются такими же):

```
Bob Smith 40000
Smith
44000.0
```

Это еще не база данных, но мы могли бы, как и прежде, вставить эти объекты в список или в словарь, чтобы объединить их в одно целое:

```
>>> from person_start import Person
>>> bob = Person('Bob Smith', 42)
>>> sue = Person('Sue Jones', 45, 40000)
>>> people = [bob, sue]        # список "базы данных"
>>> for person in people:
        print(person.name, person.pay)

Bob Smith 0
Sue Jones 40000

>>> x = [(person.name, person.pay) for person in people]
>>> x
[('Bob Smith', 0), ('Sue Jones', 40000)]
```

```
>>> [rec.name for rec in people if rec.age >= 45]  # SQL-подобный запрос
['Sue Jones']

>>> [(rec.age ** 2 if rec.age >= 45 else rec.age) for rec in people]
[42, 2025]
```

Обратите внимание, что для Боба был установлен оклад (поле `pay`) по умолчанию, равный 0, потому что при создании записи мы не указали сумму в соответствующем аргументе (может быть, Сью его поддерживает?). Мы также могли бы реализовать класс, представляющий базу данных, возможно, как подкласс списка или словаря, добавив в него методы вставки и удаления, реализующие особенности функционирования базы данных. Однако пока мы откажемся от этого, потому что гораздо полезнее реализовать сохранение записей в хранилище, которое уже обладает методами записи и чтения. Но прежде чем попытаться использовать хранилище, добавим в наши записи немного логики.

Добавляем поведение

Пока что наш класс – это всего лишь данные: он заменил ключи словаря атрибутами объекта, но не добавляет ничего нового сверх того, что у нас было прежде. Чтобы задействовать всю мощь классов, необходимо добавить реализацию поведения. Заключая реализацию поведения в методы класса, мы сможем изолировать клиентов от влияния изменений в будущем. А объединяя методы в единое целое с данными, мы обеспечиваем естественное место, где другие будут искать наш программный код. В некотором смысле классы объединяют в себе записи и программы, обрабатывающие эти записи, – методы реализуют логику интерпретации и изменения данных (этот стиль программирования потому и называется *объектно-ориентированным*, что при таком подходе всегда обрабатываются данные объектов).

Например, в примере 1.15 добавляется логика получения фамилии и увеличения оклада в виде методов. Для доступа к обрабатываемому экземпляру (записи) методы используют аргумент `self`.

Пример 1.15. PP4E\Preview\person.py

```
class Person:
    def __init__(self, name, age, pay=0, job=None):
        self.name = name
        self.age  = age
        self.pay  = pay
        self.job  = job
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith', 42, 30000, 'software')
    sue = Person('Sue Jones', 45, 40000, 'hardware')
    print(bob.name, sue.pay)

    print(bob.lastName())
    sue.giveRaise(.10)
    print(sue.pay)
```

Если запустить этот сценарий, он выведет те же результаты, что и предыдущий, но теперь эти результаты возвращаются методами, а не извлекаются с помощью жестко зашитой логики, которая всегда оказывается избыточной, когда бы она ни применялась:

```
Bob Smith 40000
Smith
44000.0
```

Добавляем наследование

И наконец, рассмотрим еще одно расширение наших записей, прежде чем перейти к реализации их сохранения: поскольку теперь записи реализованы в виде класса, они обретают естественную возможность специализации через механизм наследования в Python. Пример 1.16 специализирует предыдущий класс `Person`, предусматривая 10-процентную надбавку, добавляемую при увеличении оклада менеджеров (любые совпадения с реальными примерами из жизни являются случайными).

Пример 1.16. PP4E\Preview\manager.py

```
from person import Person

class Manager(Person):
    def giveRaise(self, percent, bonus=0.1):
        self.pay *= (1.0 + percent + bonus)

if __name__ == '__main__':
    tom = Manager(name='Tom Doe', age=50, pay=50000)
    print(tom.lastName())
    tom.giveRaise(.20)
    print(tom.pay)
```

Если запустить этот сценарий, он выведет следующее:

```
Doe
65000.0
```

Здесь объявление класса `Manager` находится в отдельном модуле, но это объявление точно так же можно поместить в модуль `person` (Python не требует создавать отдельные модули для каждого класса). Он наследует конструктор и метод `lastName` от своего суперкласса и специализирует

метод giveRaise (как будет показано позднее, существуют различные способы реализации этого расширения). Поскольку данное дополнение было оформлено в виде нового подкласса, оно никак не отразится на работе экземпляров оригинального класса Person. Экземпляры, представляющие информацию о Бобе и Сью, например, унаследуют оригинальную логику увеличения оклада, а экземпляр, представляющий информацию о Томе, получит специализированную версию, потому что он является экземпляром другого класса. В ООП программы разрабатываются за счет специализации программного кода, а не его изменения.

Фактически программа, использующая наши объекты, вообще ничего не должна знать об особенностях реализации метода giveRaise – объект сам определяет порядок выполнения той или иной операции, опираясь на класс, экземпляром которого он является. Пока объект поддерживает ожидаемый интерфейс (в данном случае метод giveRaise), он будет совместим с вызывающей программой, независимо от конкретного типа объекта и даже независимо от особенностей реализации этого метода, которая может действовать совершенно иначе.

Если вам уже приходилось изучать язык Python, возможно, вы знаете, что такое поведение называется *полиморфизмом*. Это одно из основных свойств языка, и оно объясняет значительную долю гибкости программного кода. Результат вызова метода giveRaise в следующем фрагменте зависит от того, к какому классу принадлежит обрабатываемый объект obj, – Том получит 20-процентное повышение оклада, а не 10-процентное, потому что соответствующий ему экземпляр является экземпляром специализированного класса Manager:

```
>>> from person import Person
>>> from manager import Manager

>>> bob = Person(name='Bob Smith', age=42, pay=10000)
>>> sue = Person(name='Sue Jones', age=45, pay=20000)
>>> tom = Manager(name='Tom Doe', age=55, pay=30000)
>>> db = [bob, sue, tom]

>>> for obj in db:
        obj.giveRaise(.10)     # метод по умолчанию или специализированный

>>> for obj in db:
        print(obj.lastName(), '=>', obj.pay)

Smith => 11000.0
Jones => 22000.0
Doe => 36000.0
```

Реструктуризация программного кода

Прежде чем двинуться дальше, рассмотрим еще несколько альтернативных вариантов реализации. Большинство из них подчеркивают

преимущества модели ООП в Python и рассматриваются здесь для краткого знакомства.

Расширение методов

Во-первых, обратите внимание на некоторую избыточность в примере 1.16: расчет увеличения оклада производится в двух местах (в двух классах). Мы могли бы реализовать специализированный класс `Manager`, не замещая унаследованный метод `giveRaise` новой реализацией, а *расширяя* его:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=0.1):
        Person.giveRaise(self, percent + bonus)
```

Вся хитрость заключается в непосредственном вызове версии метода суперкласса и явной передаче ему аргумента `self`. При таком подходе мы также переопределяем метод, но на этот раз мы просто вызываем универсальную версию после добавления 10-процентной надбавки (предусмотренной по умолчанию) к указанному значению в процентах. Этот прием позволяет уменьшить избыточность программного кода (оригинальная логика метода `giveRaise` находится в одном только месте, что упрощает возможность ее изменения в будущем), и его особенно удобно использовать при переопределении методов-конструкторов суперклассов.

Если вы уже знакомы с особенностями ООП в Python, то должны знать, что этот прием работает благодаря возможности вызова методов как относительно экземпляра, так и относительно имени класса. Вообще говоря, следующие два вызова являются эквивалентными, и можно использовать обе формы:

```
instance.method(arg1, arg2)
class.method(instance, arg1, arg2)
```

В действительности, первая форма отображается во вторую – при вызове метода относительно экземпляра интерпретатор Python отыскивает в дереве наследования ближайший класс, в котором имеется требуемый метод, и вызывает его, автоматически передавая экземпляр в первом аргументе. В любом случае, внутри метода `giveRaise` аргумент `self` будет ссылаться на экземпляр, являющийся объектом вызова.

Формат отображения

Чтобы получить дополнительное удовольствие от использования ООП, мы могли бы добавить в наши классы несколько методов перегрузки операторов. Например, метод `__str__`, реализованный здесь, возвращает отформатированную строку для отображения наших объектов при печати объектов целиком – такое представление выглядит гораздо лучше, чем предусмотренное по умолчанию:

```
class Person:
    def __str__(self):
        return '<%s => %s>' % (self.__class__.__name__, self.name)

tom = Manager('Tom Jones', 50)
print(tom)                    # выведет: <Manager => Tom Jones>
```

Здесь атрибут `__class__` содержит ссылку на ближайший класс, экземпляром которого является объект `self`, даже при том, что метод `__str__` может оказаться унаследованной версией. Метод `__str__` позволяет выводить экземпляры непосредственно, вместо того чтобы выводить отдельные атрибуты. В метод `__str__` можно было бы добавить цикл, выполняющий обход словаря атрибутов `__dict__` экземпляра и отображающий все атрибуты. Но это лишь краткий обзор, поэтому оставим это предложение для самостоятельного упражнения.

Мы могли бы даже реализовать метод `__add__`, чтобы оператор + автоматически вызывал метод `giveRaise`. Нужно ли это – другой вопрос. Использование оператора + для увеличения оклада может быть истолковано неправильно теми, кто впоследствии будет читать наш программный код.

Специализация конструктора

Наконец, обратите внимание, что в примере 1.16 при создании экземпляра класса `Manager` мы не передаем конструктору аргумент `job`. При необходимости мы могли бы передавать это значение в виде именованного аргумента, как показано ниже:

```
tom = Manager(name='Tom Doe', age=50, pay=50000, job='manager')
```

Причина, по которой мы в примере не включили передачу аргумента `job`, заключается в том, что в этом нет необходимости: если создается новый экземпляр класса `Manager`, занимаемая должность уже подразумевается классом. Тем не менее, чтобы не оставлять поле `job` пустым, возможно, имеет смысл явно реализовать конструктор для класса `Manager`, который будет заполнять это поле автоматически:

```
class Manager(Person):
    def __init__(self, name, age, pay):
        Person.__init__(self, name, age, pay, 'manager')
```

Теперь при создании экземпляра класса `Manager` его поле `job` будет заполняться автоматически. Вся хитрость заключается в явном вызове версии метода суперкласса, так же, как мы делали при реализации метода `giveRaise` выше. Единственное отличие здесь – необычное имя метода-конструктора.

Альтернативные классы

В последующих примерах мы не будем использовать ни одно из трех расширений, представленных в этом разделе, но для демонстрации их

в действии соберем все эти идеи в примере 1.17, где представлены альтернативные реализации классов Person и Manager.

Пример 1.17. PP4E\Preview\person_alternative.py

```
"""
Альтернативные реализации классов Person и Manager с данными, методами
и с перегрузкой операторов (не используется в объектах, предусматривающих
возможность сохранения)
"""
class Person:
    """
    универсальное представление человека: данные+логика
    """
    def __init__(self, name, age, pay=0, job=None):
        self.name = name
        self.age  = age
        self.pay  = pay
        self.job  = job
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def __str__(self):
        return ('<%s => %s: %s, %s>' %
                (self.__class__.__name__, self.name, self.job, self.pay))

class Manager(Person):
    """
    класс со специализированным методом giveRaise,
    наследующий обобщенные методы lastName и __str__
    """
    def __init__(self, name, age, pay):
        Person.__init__(self, name, age, pay, 'manager')
    def giveRaise(self, percent, bonus=0.1):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith', 44)
    sue = Person('Sue Jones', 47, 40000, 'hardware')
    tom = Manager(name='Tom Doe', age=50, pay=50000)
    print(sue, sue.pay, sue.lastName())
    for obj in (bob, sue, tom):
        obj.giveRaise(.10)     # вызовет метод giveRaise объекта obj
        print(obj)             # вызовет обобщенную версию метода __str__
```

Обратите внимание на полиморфизм в цикле for, находящемся в программном коде самопроверки этого модуля: все три объекта используют один и тот же конструктор, метод lastName и методы вывода, но при обращении к методу giveRaise вызывается версия в зависимости от класса, на основе которого был создан экземпляр. Если запустить сце-

нарий из примера 1.17, он выведет в стандартный поток вывода приведенные ниже строки; поле job в экземпляре класса Manager заполняется конструктором, форматированный вывод наших объектов осуществляется с помощью нового метода __str__, а новая версия метода giveRaise в классе Manager действует точно так же, как и прежде:

```
<Person => Sue Jones: hardware, 40000> 40000 Jones
<Person => Bob Smith: None, 0.0>
<Person => Sue Jones: hardware, 44000.0>
<Manager => Tom Doe: manager, 60000.0>
```

Такая *реструктуризация* программного кода часто применяется по мере роста и развития иерархий классов. Фактически мы никак не сможем увеличить оклад тем, у кого он оказался равным нулю (Бобу явно не повезло), поэтому нам, вероятно, необходимо предусмотреть возможность прямого изменения оклада, но оставим это усовершенствование до следующей версии. Самое приятное, что гибкость и удобочитаемость, присущие языку Python, существенно упрощают реструктуризацию программного кода – вы легко и просто сможете реструктурировать свои программы. Если прежде вы не пользовались языком Python, то со временем обнаружите, что разработка программ на Python выполняется быстро, поэтапно и в интерактивном режиме, что хорошо подходит для постоянно изменяющихся потребностей реальных проектов.

Добавляем возможность сохранения

Пришло время продолжить. Теперь у нас имеются реализации записей, поддающиеся специализации и включающие логику их обработки, в форме классов. Осталось сделать последний маленький шаг и реализовать сохранение наших записей, основанных на классах. Мы могли бы снова сохранять каждую запись в отдельном файле с помощью модуля pickle, но модуль shelve предоставляет точно такую же возможность, а кроме того, его гораздо проще использовать. Как это сделать, демонстрируется в примере 1.18.

Пример 1.18. PP4E\Preview\make_db_classes.py

```
import shelve
from person import Person
from manager import Manager

bob = Person('Bob Smith', 42, 30000, 'software')
sue = Person('Sue Jones', 45, 40000, 'hardware')
tom = Manager('Tom Doe', 50, 50000)

db = shelve.open('class-shelve')
db['bob'] = bob
db['sue'] = sue
db['tom'] = tom
db.close()
```

Этот сценарий создает три экземпляра (два экземпляра оригинального класса и один – его специализированной версии) и присваивает их ключам вновь созданного хранилища. Другими словами, сценарий создает хранилище с экземплярами классов. Для нашего программного кода база выглядит в точности, как словарь экземпляров классов, с той лишь разницей, что словарь верхнего уровня отображается в файл хранилища, как и прежде. Убедиться, что все работает, поможет сценарий в примере 1.19, который читает содержимое хранилища и выводит значения полей записей.

Пример 1.19. PP4E\Preview\dump_db_classes.py

```
import shelve
db = shelve.open('class-shelve')
for key in db:
    print(key, '=>\n ', db[key].name, db[key].pay)

bob = db['bob']
print(bob.lastName())
print(db['tom'].lastName())
```

Обратите внимание, что в этом примере нам не требуется импортировать класс `Person`, чтобы извлекать экземпляры из хранилища или вызывать их методы. Когда экземпляры сохраняются с помощью модуля `shelve` или `pickle`, используемая этими модулями система сохранения записывает в файл не только значения атрибутов экземпляров, но и дополнительную информацию, позволяющую позднее автоматически определить местоположение классов при извлечении экземпляров (модули с определениями классов просто должны находиться в пути поиска модулей при выполнении операции загрузки). Это сделано специально, потому что определение класса и его экземпляры в хранилище сохраняются отдельно; вы можете изменить класс, чтобы изменить порядок интерпретации экземпляров при загрузке (подробнее об этом рассказывается далее в книге). Ниже приводятся результаты запуска сценария *dump_db_classes.py* сразу после создания хранилища с помощью сценария *make_db_classes.py*:

```
bob =>
   Bob Smith 30000
sue =>
   Sue Jones 40000
tom =>
   Tom Doe 50000
Smith
Doe
```

Как показано в примере 1.20, изменение информации в базе данных выполняется так же просто, как и прежде (сравните с примером 1.13), но на этот раз вместо ключей словарей используются атрибуты экземпляров, а жестко зашитую логику изменений заменили вызовы мето-

дов классов. Обратите внимание, что нам по-прежнему необходимо извлечь запись, изменить ее и вновь присвоить тому же самому ключу.

Пример 1.20. PP4E\Preview\update_db_classes.py

```
import shelve
db = shelve.open('class-shelve')

sue = db['sue']
sue.giveRaise(.25)
db['sue'] = sue

tom = db['tom']
tom.giveRaise(.20)
db['tom'] = tom
db.close()
```

И наконец, ниже приводятся результаты повторного запуска сценария *dump_db_classes.py* после запуска сценария *update_db_classes.py*. Том и Сью теперь имеют новые оклады, потому что теперь соответствующие объекты сохраняются в хранилище. Кроме того, мы могли бы открыть и исследовать содержимое хранилища в интерактивной оболочке Python – несмотря на свою долговечность, хранилище является всего лишь объектом Python, содержащим другие объекты Python.

```
bob =>
   Bob Smith 30000
sue =>
   Sue Jones 50000.0
tom =>
   Tom Doe 65000.0
Smith
Doe
```

Том и Сью получили прибавку к окладу, потому что теперь эти объекты – объекты, сохраненные в базе данных. Хотя модуль `shelve` также способен сохранять объекты более простых типов, таких как списки и словари, однако классы позволяют нам объединять данные и поведение в единые сохраняемые элементы. В некотором смысле атрибуты экземпляров и методы классов равносильны записям и обрабатывающим их программам, используемым в более традиционных решениях.

Другие разновидности баз данных

К настоящему моменту мы создали вполне функциональную базу данных: наши классы одновременно реализуют хранение данных записей и их обработку и заключают в себе реализацию поведения. А модули `pickle` и `shelve` обеспечивают простой способ сохранения нашей базы данных между запусками программы. Это не реляционная база данных (она хранит объекты, а не таблицы, и запросы имеют вид программного

кода на языке Python, обрабатывающего объекты), но ее вполне достаточно для многих видов программ.

Если потребуются более широкие функциональные возможности, мы сможем перевести это приложение на использование более мощных инструментов. Например, если нам потребуется полноценная поддержка запросов на языке SQL, мы сможем использовать библиотеки, позволяющие сценариям на языке Python путем переноса взаимодействовать с реляционными базами данных, такими как MySQL, PostgreSQL и Oracle.

Механизмы ORM (Object Relational Mapper – объектно-реляционное отображение), такие как SQLObject и SqlAlchemy, предлагают иной подход, сохраняющий представление записей в виде объектов Python, но преобразуя их в и из представления таблиц в реляционных базах данных, в некотором смысле обеспечивая сочетание лучших черт обоих миров – с синтаксисом классов Python сверху и надежными базами данных внутри.

Кроме того, существует открытая система ZODB, реализующая более функциональную объектную базу данных для программ на языке Python, с поддержкой особенностей, отсутствующих в хранилищах `shelve`, включая параллельное изменение записей, подтверждение и откат транзакций, автоматическое обновление компонентов, изменившихся в оперативной памяти, и многие другие. Мы познакомимся с этими, более совершенными инструментами, созданными сторонними разработчиками, в главе 17. А теперь перейдем к созданию лица нашей системы.

Автобусы признаны опасными

На протяжении многих лет Python пользовался мощной и добровольной поддержкой и отдельных лиц, и организаций. В настоящее время конференции и другие некоммерческие мероприятия в сообществе Python проходят при содействии некоммерческой организации Python Software Foundation (PSF). Организации PSF предшествовала организация PSA – группа, которая первоначально была образована в ответ на когда-то давно возникшее в телеконференции Python обсуждение полусерьезного вопроса: «Что будет, если Гвидо попадет под автобус?»

В настоящее время создатель языка Python, Гвидо ван Россум (Guido van Rossum), по-прежнему является верховным арбитром при поступлении предложений о внесении изменений в Python. Он официально был помазан на пост Великодушного Пожизненного Диктатора (Benevolent Dictator for Life, BDFL) на первой же конференции Python, и по-прежнему окончательное решение о принятии

изменений в языке остается за ним (и многие изменения, ведущие к несовместимости, за исключением версии 3.0, несовместимость которой была предусмотрена заранее, он обычно отклоняет: это хорошая черта для языков программирования, потому что Python должен изменяться достаточно медленно и изменения не должны нарушать обратную совместимость).

Но, как бы то ни было, огромное количество пользователей Python помогает поддерживать язык, работает над расширениями, исправляет ошибки и так далее. Это по-настоящему совместный проект. Фактически разработка Python сейчас является совершенно открытым процессом – любой желающий сможет получить самые свежие файлы с исходными текстами или отправить свои исправления, посетив веб-сайт проекта (подробности вы найдете по адресу *http://www.python.org*).

Разработка Python, как пакета открытого программного обеспечения, в действительности находится в руках большого количества программистов, разбросанных по всему свету. Поэтому, даже если Великодушный Пожизненный Диктатор когда-нибудь передаст факел преемнику, Python практически наверняка продолжит пользоваться поддержкой своих пользователей. По своей природе открытые проекты, хотя и не без отступлений, обычно отражают потребности сообществ их пользователей в большей степени, чем потребности отдельных личностей или учредителей.

С учетом популярности Python нападение со стороны автобуса уже не кажется таким опасным, как раньше. Впрочем, Гвидо может считать иначе.

Шаг 4: добавляем интерфейс командной строки

Пока что наша программа состоит из экземпляров классов, реализованных в предыдущем разделе, которые сохранены в файле хранилища. В качестве хранилища она делает достаточно, но для доступа к содержимому и работы с ним нам потребуется запускать другие сценарии или вводить программный код в интерактивной оболочке. Улучшить ситуацию достаточно просто: необходимо написать универсальную программу, которая будет взаимодействовать с пользователем либо с помощью окна консоли, либо с помощью графического интерфейса.

Интерфейс командной строки к хранилищу

Начнем с наиболее простого варианта. В самом простом виде интерфейс к базе данных должен давать пользователям возможность вводить ключи и значения в окне консоли (вместо того чтобы писать программный

код на языке Python). Сценарий в примере 1.21 реализует простейший цикл интерактивных взаимодействий, позволяя пользователю запрашивать объекты, имеющиеся в хранилище.

Пример 1.21. PP4E\Preview\peopleinteract_query.py

```
# интерактивные запросы
import shelve
fieldnames = ('name', 'age', 'job', 'pay')
maxfield   = max(len(f) for f in fieldnames)
db = shelve.open('class-shelve')

while True:
    key = input('\nKey? => ')  # ключ или пустая строка, возбуждает исключение
                               # при вводе EOF
    if not key: break
        try:
            record = db[key]   # извлечь запись по ключу и вывести
        except:
            print('No such key "%s"!' % key)
        else:
            for field in fieldnames:
                print(field.ljust(maxfield), '=>', getattr(record, field))
```

Для извлечения значений атрибутов в этом сценарии используется встроенная функция `getattr`, а для форматирования вывода используется строковый метод `ljust`, выравнивающий строку по левому краю (значение `maxfield`, порожденное выражением-генератором, представляет длину наибольшего имени поля). После запуска сценария он входит в цикл, предлагает пользователю ввести ключ (со стандартного потока ввода, который обычно соответствует окну консоли) и отображает извлеченную запись поле за полем. Ввод пустой строки завершает сеанс работы со сценарием. Предположим, хранилище находится в том же состоянии, в каком мы его оставили ближе к концу предыдущего раздела:

```
...\PP4E\Preview> dump_db_classes.py
bob =>
   Bob Smith 30000
sue =>
   Sue Jones 50000.0
tom =>
   Tom Doe 65000.0
Smith
Doe
```

Мы сможем использовать наш новый сценарий для запроса объектов из базы данных в интерактивном режиме:

```
...\PP4E\Preview> peopleinteract_query.py
Key? => sue
name => Sue Jones
```

```
age  => 45
job  => hardware
pay  => 50000.0

Key? => nobody
No such key "nobody"!

Key? =>
```

Сценарий в примере 1.22 является следующим шагом к созданию интерфейса, позволяющим вносить изменения в интерактивном режиме. Для заданного ключа он позволяет ввести значения для каждого из полей и либо изменяет существующую запись, либо создает новую, после чего сохраняет ее с указанным ключом.

Пример 1.22. PP4E\Preview\peopleinteract_update.py

```
# интерактивные изменения
import shelve
from person import Person
fieldnames = ('name', 'age', 'job', 'pay')

db = shelve.open('class-shelve')
while True:
    key = input('\nKey? => ')
    if not key: break
    if key in db:
        record = db[key]                    # изменить существующую
    else:                                   # или создать новую запись
        record = Person(name='?', age='?') # для eval: строки в кавычках
    for field in fieldnames:
        currval = getattr(record, field)
        newtext = input('\t[%s]=%s\n\t\tnew?=>' % (field, currval))
        if newtext:
            setattr(record, field, eval(newtext))
    db[key] = record
db.close()
```

Обратите внимание, что для преобразования введенных значений в этом сценарии используется функция `eval` (это позволяет вводить любые объекты Python, но это означает, что строки при вводе должны заключаться в кавычки). Функция `setattr` присваивает значение атрибуту, имя которого задается строкой. Этот сценарий позволит добавлять или изменять любое количество записей – чтобы сохранить прежнее значение поля в записи, достаточно просто нажать клавишу Enter в ответ на просьбу ввести новое значение:

```
Key? => tom
        [name]=Tom Doe
                new?=>
        [age]=50
                new?=>56
```

```
        [job]=None
                new?=>'mgr'
        [pay]=65000.0
                new?=>90000

Key? => nobody
        [name]=?
                new?=>'John Doh'
        [age]=?
                new?=>55
        [job]=None
                new?=>
        [pay]=0
                new?=>None

Key? =>
```

Этот сценарий все еще очень прост (в нем, например, не предусмотрена обработка ошибок), но пользоваться им гораздо удобнее, чем вручную открывать и вносить изменения в хранилище в интерактивной оболочке Python, особенно для тех, кто не занимается программированием. Запустим сценарий *peopleinteract_query.py*, чтобы проверить изменения, которые мы внесли (если кому-то такой подход покажется утомительным, он сможет объединить оба сценария в один, ценой дополнительного программного кода и повышения сложности для пользователя):

```
Key? => tom
name => Tom Doe
age  => 56
job  => mgr
pay  => 90000

Key? => nobody
name => John Doh
age  => 55
job  => None
pay  => None

Key? =>
```

Шаг 5: добавляем графический интерфейс

Интерфейс командной строки, реализованный в предыдущем разделе, вполне работоспособен, и для некоторых пользователей его вполне достаточно, если предположить, что им не досаждает ввод команд в окне консоли. Однако, приложив небольшое количество усилий, мы сможем добавить современный графический интерфейс, более простой в использовании, снижающий риск ошибок и определенно более привлекательный.

Основы графических интерфейсов

Как будет показано далее в этой книге, программистам, использующим язык Python, доступно множество разнообразных инструментов создания графических интерфейсов: tkinter, wxPython, PyQt, PythonCard, Dabo и многие другие. Из них в составе Python поставляется только tkinter, который де-факто считается стандартным инструментом.

tkinter – это легковесный инструмент, который прекрасно интегрируется с языками сценариев, такими как Python. Его легко использовать для реализации простых графических интерфейсов, а дополнительные расширения к нему и применение приемов объектно-ориентированного программирования позволяют без особых затрат реализовать более сложные интерфейсы. Кроме того, реализация графического интерфейса на базе tkinter способна без каких-либо модификаций работать в Windows, Linux/Unix и Macintosh – достаточно просто перенести файлы с исходными текстами на компьютер, где предполагается использовать программу с графическим интерфейсом. В tkinter отсутствуют разнообразные «бантики и рюшечки», имеющиеся в более развитых инструментах, таких как wxPython или PyQt, но это же является основной причиной его относительной простоты, что делает его идеальным инструментом для тех, кто только начинает создавать программы с графическим интерфейсом.

Инструмент tkinter разработан для использования в сценариях, поэтому программирование графических интерфейсов с его применением организовано с достаточной очевидностью. Далее в этой книге мы поближе познакомимся со всеми его понятиями и возможностями. А в качестве первого примера рассмотрим программу, использующую tkinter, которая содержит всего несколько строк программного кода, как видно из примера 1.23.

Пример 1.23. PP4E\Preview\tkinter001.py

```
from tkinter import *
Label(text='Spam').pack()
mainloop()
```

Импортировав модуль `tkinter` (на самом деле, в Python 3 – пакет модулей), мы получаем возможность обращаться к различным экранным конструкциям (или «виджетам»), таким как `Label`; методам менеджера геометрии, таким как `pack`; предварительно установленным комплектам настроек виджетов, таким как `TOP` и `RIGHT`, определяющим край для выравнивания компонентов и используемых при вызове метода `pack`; и к функции `mainloop`, запускающей цикл обработки событий.

Это не самый полезный сценарий с графическим интерфейсом из когда-либо созданных, но он демонстрирует основы использования tkinter и создает полнофункциональное окно, как показано на рис. 1.1, – всего тремя строками программного кода. Изображение окна, как

и всех других графических интерфейсов в этой книге, было получено в Windows 7. Окно действует одинаково и на других платформах (таких как Mac OS X, Linux и в более старых версиях Windows), но при этом имеет внешний вид, характерный для той платформы, на которой запускается сценарий.

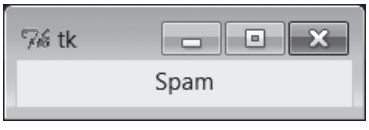


***Рис. 1.1.** Окно сценария tkinter001.py*

Вы можете запустить этот сценарий из среды IDLE, из командной строки или щелчком на ярлыке – то есть точно так же, как любой другой сценарий на языке Python. Сам инструмент tkinter является стандартной частью Python и работает, что называется, «из коробки», на всех платформах, включая Windows. Однако в некоторых операционных системах может потребоваться выполнить дополнительные шаги по установке и настройке (подробнее об этом будет рассказываться далее в книге).

Совсем немного усилий требуется приложить, чтобы создать графический интерфейс, откликающийся на действия пользователя: сценарий в примере 1.24 реализует графический интерфейс с кнопкой, щелчок на которой приводит к вызову функции `reply`.

Пример 1.24. PP4E\Preview\ tkinter101.py

```
from tkinter import *
from tkinter.messagebox import showinfo

def reply():
    showinfo(title='popup', message='Button pressed!')

window = Tk()
button = Button(window, text='press', command=reply)
button.pack()
window.mainloop()
```

Этот пример также достаточно прост. Он явно создает главное окно `Tk` приложения, которое будет служить контейнером для кнопки, и воспроизводит на экране простое окно, как показано на рис. 1.2 (при создании нового виджета в модуле `tkinter` принято передавать контейнерные элементы в первом аргументе; который по умолчанию ссылается на главное окно). Но на этот раз при каждом щелчке на кнопке с надписью `press` программа будет откликаться вызовом программного кода, который выводит диалог, как показано на рис. 1.3.

Рис. 1.2. *Главное окно сценария tkinter101.py*

Обратите внимание, что окно диалога выглядит в Windows 7 (на платформе, где сделан снимок с экрана) именно так, как и должно выглядеть. Повторюсь, что модуль `tkinter` обеспечивает внешний вид создаваемых элементов, характерный для той платформы, на которой запускается сценарий. Мы можем изменять самые разные аспекты графического интерфейса (например, цвет и шрифт, текст и ярлык в заголовке окна, помещать на кнопки изображения вместо текста), но одно из преимуществ использования модуля `tkinter` – в том, что нам необходимо будет изменять лишь те параметры, которые требуется изменить.

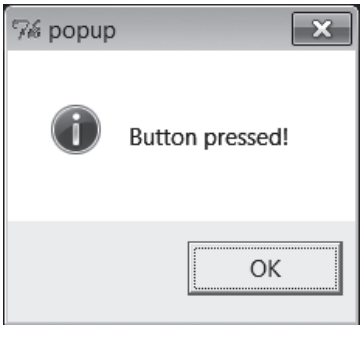


Рис. 1.3. *Типичное окно диалога, созданное сценарием tkinter101.py*

ООП при разработке графических интерфейсов

Все примеры реализации графического интерфейса, представленные до сих пор, имели вид самостоятельных сценариев, включающих функцию обработки событий. В крупных программах часто более полезно бывает оформить программный код, создающий графический интерфейс, в виде подкласса виджета `Frame` из библиотеки tkinter – контейнера для других виджетов. В примере 1.25 приводится переработанная версия предыдущего сценария с одной кнопкой, в которой графический интерфейс реализован в виде класса.

Пример 1.25. PP4E\Preview\tkinter102.py

```
from tkinter import *
from tkinter.messagebox import showinfo
```

```
class MyGui(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        button = Button(self, text='press', command=self.reply)
        button.pack()
    def reply(self):
        showinfo(title='popup', message='Button pressed!')

if __name__ == '__main__':
    window = MyGui()
    window.pack()
    window.mainloop()
```

Обработчик событий от кнопки – это *связанный метод* `self.reply`, то есть объект, хранящий в себе значение `self` и ссылку на метод `reply`. Данный пример воспроизводит то же самое окно и диалог, что и сценарий в примере 1.24 (рис. 1.2 и 1.3). Но теперь графический интерфейс реализован как подкласс класса `Frame` и потому автоматически становится присоединяемым *компонентом* – то есть мы сможем добавить все виджеты, создаваемые этим классом, как единый пакет в любой другой графический интерфейс; достаточно просто присоединить экземпляр этого класса к графическому интерфейсу. Как это делается, показано в примере 1.26.

Пример 1.26. PP4E\Preview\attachgui.py

```
from tkinter import *
from tkinter102 import MyGui

# главное окно приложения
mainwin = Tk()
Label(mainwin, text=__name__).pack()

# окно диалога
popup = Toplevel()
Label(popup, text='Attach').pack(side=LEFT)
MyGui(popup).pack(side=RIGHT)                  # присоединить виджеты
mainwin.mainloop()
```

Этот сценарий присоединяет наш графический интерфейс с одной кнопкой к другому окну `popup` типа `Toplevel`, которое передается импортированному приложению через вызов конструктора, как родительский компонент (кроме того, вы получаете доступ к главному окну `Tk` – как будет показано позже, вы всегда сможете получить к нему доступ, независимо от того, создается оно явно или нет). На этот раз наш пакет виджетов, содержащий единственную кнопку, присоединяется к правому краю контейнера. Если запустить этот пример, вы увидите картину, изображенную на рис. 1.4, где кнопка с надписью press – это наш подкласс класса `Frame`.

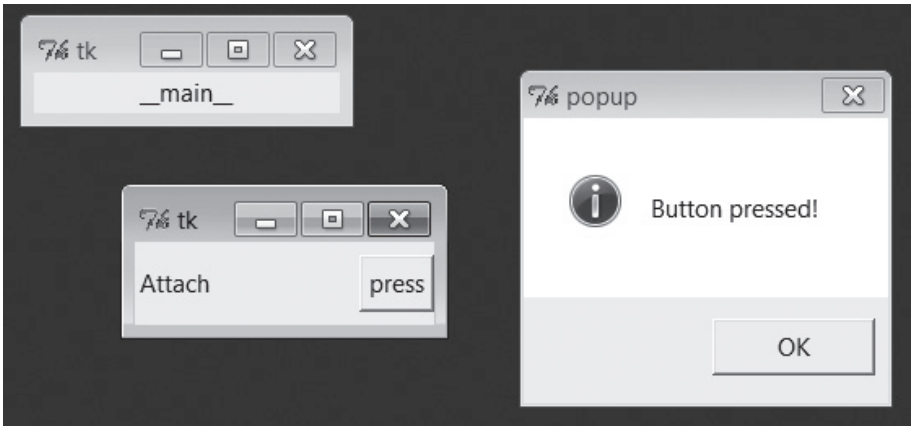


***Рис. 1.4.** Присоединение интерфейсных элементов*

Кроме того, так как наш графический интерфейс `MyGui` оформлен в виде класса, мы получаем возможность настраивать его за счет использования механизма наследования – достаточно просто определить подкласс, реализующий необходимые отличия. Например, можно изменить реализацию метода `reply`, чтобы он делал что-то особенное, как показано в примере 1.27.

Пример 1.27. PP4E\Preview\customizegui.py

```
from tkinter import mainloop
from tkinter.messagebox import showinfo
from tkinter102 import MyGui

class CustomGui(MyGui):                              # наследует метод __init__
    def reply(self):                                 # замещает метод reply
        showinfo(title='popup', message='Ouch!')

if __name__ == '__main__':
    CustomGui().pack()
    mainloop()
```

Если запустить этот сценарий, он создаст то же главное окно с кнопкой, что и оригинальный класс `MyGui`. Но щелчок на кнопке сгенерирует иной ответ, как показано на рис. 1.5, потому что будет вызвана другая версия метода `reply`.

Несмотря на свою простоту, эти графические интерфейсы иллюстрируют несколько важных идей. Как будет показано далее в этой книге, использование приемов ООП, таких как наследование и присоединение в данных примерах, позволяет повторно использовать пакеты виджетов в наших программах – калькуляторы, текстовые редакторы и подобные им интерфейсы легко могут настраиваться и добавляться в дру-

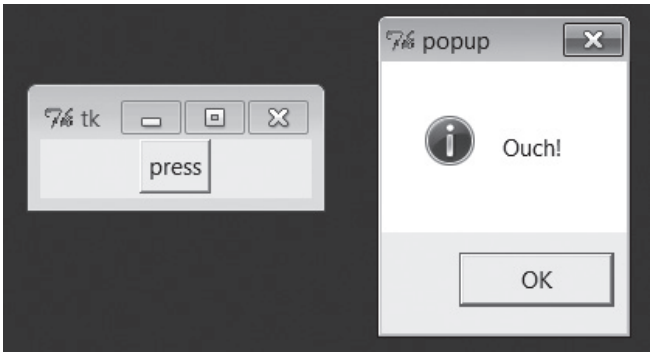


***Рис. 1.5.** Изменение графического интерфейса*

гие графические интерфейсы как компоненты, если они реализованы в виде классов. Кроме того, подклассы виджетов способны обеспечить единство внешнего вида или стандартизованное поведение всех своих экземпляров, что в иных терминах может быть названо стилями и темами графического интерфейса. Это обычный побочный продукт Python и ООП.

Получение ввода от пользователя

В примере 1.28 приводится заключительный пример вводного сценария, демонстрирующий, как получить ввод пользователя с помощью виджета `Entry` и вывести его в диалоге. Использованная здесь инструкция `lambda` откладывает вызов функции `reply` до момента, когда ввод пользователя можно будет передать дальше, – это типичный прием программирования при работе с модулем `tkinter`. Без инструкции `lambda` функция `reply` была бы вызвана в момент создания кнопки, а не в момент щелчка на ней (мы могли бы использовать глобальную переменную `ent` внутри функции `reply`, но это делает функцию менее универсальной). Кроме того, этот пример демонстрирует, как изменить ярлык и текст в заголовке окна верхнего уровня. В данном случае файл ярлыка находится в том же каталоге, что и сценарий (если в вашей системе вызов метода `iconbitmap` терпит неудачу, попробуйте закомментировать этот вызов – к сожалению, в разных платформах работа с ярлыками выполняется по-разному).

Пример 1.28. PP4E\Preview\tkinter103.py

```
from tkinter import *
from tkinter.messagebox import showinfo

def reply(name):
    showinfo(title='Reply', message='Hello %s!' % name)
```

```
top = Tk()
top.title('Echo')
top.iconbitmap('py-blue-trans-out.ico')

Label(top, text="Enter your name:").pack(side=TOP)
ent = Entry(top)
ent.pack(side=TOP)
btn = Button(top, text="Submit", command=(lambda: reply(ent.get())))
btn.pack(side=LEFT)

top.mainloop()
```

В этом примере к главному окну `Tk` присоединяются всего три виджета. Далее вы узнаете, как использовать вложенные контейнеры `Frame` виджетов для достижения различных схем размещения этих трех виджетов. На рис. 1.6 изображены главное окно и окно диалога, появляющееся после щелчка на кнопке Submit. Нечто похожее мы увидим далее в этой главе, но реализованное на языке разметки HTML – для отображения в веб-броузере.

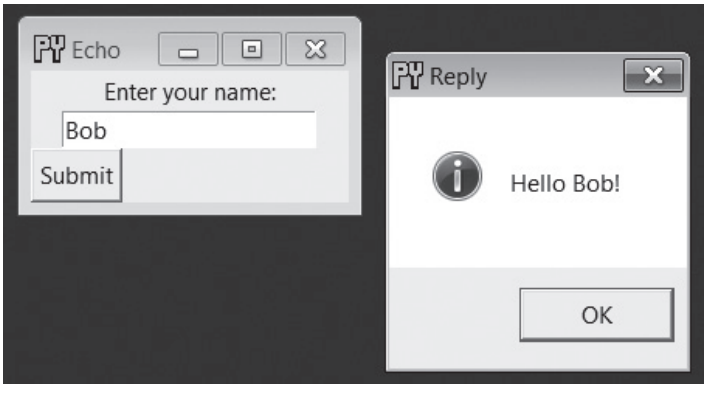


Рис. 1.6. *Получение ввода пользователя*

Программный код, представленный выше, демонстрирует множество особенностей программирования графических интерфейсов, но модуль `tkinter` обладает намного более широкими возможностями, чем можно было бы заключить из этих примеров. В модуле `tkinter` реализованы более 20 виджетов и еще множество способов дать пользователю возможность вводить данные, включая элементы ввода многострочного текста, «холсты» для рисования, раскрывающиеся меню, радиокнопки и флажки, полосы прокрутки, а также механизмы управления размещением виджетов и обработки событий. Помимо модуля `tkinter` в состав стандартной библиотеки языка Python входят также расширения, такие как PMW, и инструменты Tix и ttk, которые добавляют допол-

нительные виджеты для использования в графических интерфейсах, построенных на базе tkinter, и позволяющие придать интерфейсу еще более профессиональный внешний вид. Чтобы в общих чертах продемонстрировать имеющиеся возможности, давайте задействуем модуль `tkinter` для создания интерфейса к нашей базе данных.

Графический интерфейс к хранилищу

Первое, что необходимо сделать для нашего приложения баз данных, – это создать графический интерфейс для просмотра хранящихся данных (форму с именами и значениями полей) и реализовать способ извлечения записей по ключу. Также было бы полезно иметь возможность изменять значения полей в записях и добавлять новые записи, заполняя пустую форму. Для простоты мы реализуем единый графический интерфейс, позволяющий решать все эти задачи. На рис. 1.7 изображено окно, которое мы создадим, отображенное в Windows 7, с содержимым записи, полученной по ключу `sue` (здесь снова используется хранилище в том состоянии, в каком мы его оставили в последний раз). Данная запись в действительности является экземпляром нашего класса, сохраненным в файле хранилища, но пользователю это должно быть безразлично.

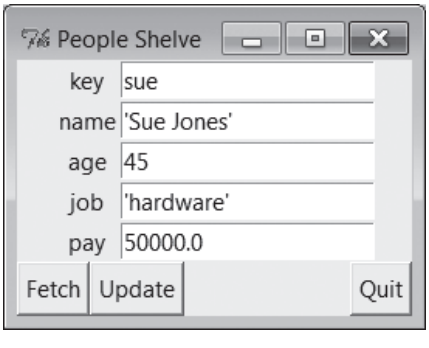


Рис. 1.7. *Главное окно сценария peoplegui.py*

Реализация графического интерфейса

Кроме того, чтобы не усложнять пример, допустим, что все записи в базе данных имеют один и тот же набор полей. Было бы совсем несложно создать более универсальную реализацию, способную работать с любыми наборами полей (и при этом создать универсальный инструмент конструирования форм с графическим интерфейсом), но мы отложим реализацию до следующих глав в этой книге. Сценарий в примере 1.29 реализует графический интерфейс, изображенный на рис. 1.7.

Пример 1.29. PP4E\Preview\peoplegui.py

```
"""
Реализация графического интерфейса для просмотра и изменения экземпляров класса,
хранящихся в хранилище;
хранилище находится на том же компьютере, где выполняется сценарий в виде одного
или более локальных файлов;
"""
from tkinter import *
from tkinter.messagebox import showerror
import shelve
shelvename = 'class-shelve'
fieldnames = ('name', 'age', 'job', 'pay')

def makeWidgets():
    global entries
    window = Tk()
    window.title('People Shelve')
    form = Frame(window)
    form.pack()
    entries = {}
    for (ix, label) in enumerate(('key',) + fieldnames):
        lab = Label(form, text=label)
        ent = Entry(form)
        lab.grid(row=ix, column=0)
        ent.grid(row=ix, column=1)
        entries[label] = ent
    Button(window, text="Fetch", command=fetchRecord).pack(side=LEFT)
    Button(window, text="Update", command=updateRecord).pack(side=LEFT)
    Button(window, text="Quit", command=window.quit).pack(side=RIGHT)
    return window

def fetchRecord():
    key = entries['key'].get()
    try:
        record = db[key]       # извлечь запись по ключу, отобразить в форме
    except:
        showerror(title='Error', message='No such key!')
    else:
        for field in fieldnames:
            entries[field].delete(0, END)
            entries[field].insert(0, repr(getattr(record, field)))

def updateRecord():
    key = entries['key'].get()
    if key in db:
        record = db[key]           # изменяется существующая запись
    else:
        from person import Person  # создать/сохранить новую запись
```

```
            record = Person(name='?', age='?') # eval: строки должны
                                               # заключаться в кавычки
        for field in fieldnames:
            setattr(record, field, eval(entries[field].get()))
        db[key] = record

db = shelve.open(shelvename)
window = makeWidgets()
window.mainloop()
db.close() # в эту точку программа попадает при щелчке на кнопке Quit
           # или при закрытии окна
```

Для размещения надписей и полей ввода в этом сценарии вместо метода `pack` используется метод `grid`. Как мы увидим далее, этот метод располагает виджеты по столбцам и строкам сетки, что обеспечивает более естественное для форм выравнивание надписей и полей ввода по горизонтали. Далее мы также увидим, что неплохого размещения виджетов на форме можно добиться и с помощью метода `pack`, добавив вложенные контейнеры для размещения виджетов по горизонтали и надписи фиксированной длины. Пока графический интерфейс никак не обрабатывает изменение размеров окна (для этого необходимы параметры настройки, которые мы будем исследовать позже), тем не менее объем программного кода реализации такой возможности при использовании любого из методов, `grid` или `pack`, будет примерно один и тот же.

Обратите внимание, что в конце сценария сначала открывается хранилище, как глобальная переменная, а затем запускается графический интерфейс – хранилище остается открытым на протяжении всего времени работы графического интерфейса (функция `mainloop` возвращает управление только после закрытия главного окна). Как будет показано в следующем разделе, такое удержание хранилища в открытом состоянии существенно отличает графический интерфейс от веб-интерфейса, где каждая операция обычно является автономной программой. Обратите также внимание, что использование глобальных переменных делает программный код более простым, но непригодным для использования вне контекста нашей базы данных; подробнее об этом мы поговорим ниже.

Пользование графическим интерфейсом

Построенный нами графический интерфейс достаточно прост, но он позволяет просматривать и изменять содержимое файла хранилища без ввода программного кода. Чтобы извлечь запись из хранилища и отобразить ее в графическом интерфейсе, необходимо ввести ключ в поле `key` (ключ) и щелкнуть на кнопке `Fetch` (Извлечь). Чтобы изменить запись, необходимо изменить содержимое полей записи после ее извлечения из хранилища и щелкнуть на кнопке `Update` (Изменить) – значения из полей ввода будут сохранены в базе данных. А чтобы добавить новую запись, необходимо заполнить все поля ввода новыми значениями

и щелкнуть на кнопке Update (Изменить) – в хранилище будет добавлена новая запись с указанным ключом и значениями полей.

Другими словами, поля ввода служат одновременно и для отображения, и для ввода. На рис. 1.8 изображена форма после добавления новой записи (щелчком на кнопке Update (Изменить)), а на рис. 1.9 – диалог с сообщением об ошибке, когда пользователь попытался извлечь запись с ключом, отсутствующим в хранилище.

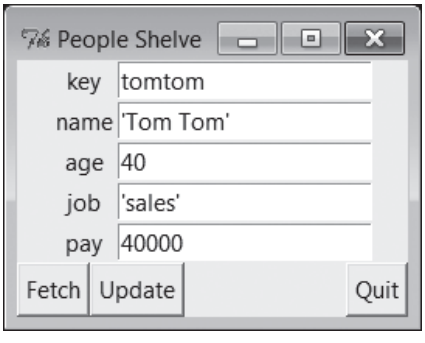


Рис. 1.8. *Интерфейс peoplegui.py после добавления нового объекта*

Рис. 1.9. *Диалог peoplegui.py с сообщением об ошибке*

Обратите внимание, что для отображения значений полей, извлеченных из хранилища, здесь также используется функция `repr`, а для преобразования значений полей в объекты Python, перед тем как они будут записаны в хранилище, – функция `eval`. Как уже упоминалось выше, это потенциально опасно, так как открывает возможность ввести в поля ввода злонамеренный программный код, но мы пока не будем касаться этой проблемы.

Однако имейте в виду, что при такой реализации строковые значения, вводимые в поля ввода, должны заключаться в кавычки, так как содержимое всех полей ввода, кроме поля key (ключ), интерпретируется как программный код на языке Python. Фактически, чтобы определить

новое значение, в поле ввода можно ввести произвольное выражение на языке Python. Например, если в поле name (имя) ввести выражение "Tom"*3, после щелчка на кнопке Update (Изменить) в записи будет сохранено имя TomTomTom. Чтобы убедиться в этом – извлеките запись из хранилища.

Несмотря на наличие графического интерфейса, позволяющего просматривать и изменять записи, мы по-прежнему можем проконтролировать свои действия, открыв и проверив файл хранилища в интерактивном режиме или запустив сценарий, такой как *dump_db_classes.py*, представленный в примере 1.19. Не забывайте: несмотря на то, что теперь мы можем просматривать записи с помощью графического интерфейса, сама база данных является файлом хранилища, содержащим объекты Python экземпляров классов, поэтому обращаться к ним может любой программный код на языке Python. Ниже приводятся результаты запуска сценария *dump_db_classes.py* после изменения существующих и добавления новых объектов с помощью графического интерфейса:

```
...\PP4E\Preview> python dump_db_classes.py
sue =>
   Sue Jones 50000.0
bill =>
   bill 9999
nobody =>
   John Doh None
tomtom =>
   Tom Tom 40000
tom =>
   Tom Doe 90000
bob =>
   Bob Smith 30000
peg =>
   1 4
Smith
Doe
```

Пути усовершенствования

Построенный нами графический интерфейс справляется со своей задачей, тем не менее в него можно внести множество разных усовершенствований:

- На настоящий момент графический интерфейс представляет собой набор функций, использующих глобальный список полей (entries) ввода и глобальное хранилище (db). Вместо этого мы могли бы передать db в вызов функции makeWidgets и организовать передачу обоих этих объектов обработчикам событий в виде аргументов, воспользовавшись приемом с инструкцией lambda из предыдущего раздела. Хотя для таких маленьких сценариев это и не так важно, стоит

иметь в виду, что явное определение подобных внешних зависимостей делает программный код более простым для понимания и повторного использования в других контекстах.

- Этот графический интерфейс можно было бы реализовать в виде класса, чтобы обеспечить поддержку возможности специализации и присоединения к другим графическим интерфейсам (глобальные переменные в этом случае могли бы стать атрибутами экземпляра), хотя повторное использование столь специфического интерфейса маловероятно.
- Полезнее было бы передавать функциям в виде параметра кортеж `fieldnames`, чтобы в будущем их можно было использовать с другими типами записей. Программный код в конце сценария также можно было бы оформить в виде функции, принимающей имя файла хранилища, а в функцию `updateRecord` можно было бы передавать функцию, создающую новую запись, чтобы она могла сохранять не только экземпляры класса `Person`. Эти усовершенствования выходят далеко за рамки данного краткого обзора, но их реализация была бы для вас неплохим упражнением. Позднее я познакомлю вас с еще одним дополнительным примером, входящим в комплект примеров к книге, PyForm, в котором используется иной подход к созданию универсальных форм ввода.
- Чтобы сделать этот графический интерфейс более дружественным по отношению к пользователю, можно было бы добавить окно со списком всех ключей, имеющихся в базе данных, и тем самым упростить просмотр содержимого базы данных. Полезно было бы предусмотреть проверку данных перед сохранением, а кроме того, легко можно было бы добавить клавиши `Delete` (Удалить) и `Clear` (Очистить). Тот факт, что введенные данные интерпретируются как программный код на языке Python, может доставить массу беспокойств – реализация простейшей схемы ввода могла бы повысить безопасность. (Я не буду явно предлагать реализовать эти усовершенствования в качестве самостоятельного упражнения, но это было бы полезно.)
- Мы могли бы также реализовать поддержку изменения размеров окна (как мы узнаем позднее, виджеты могут растягиваться и сжиматься вместе с окном) и предоставить возможность вызова методов, которыми обладают сохраняемые экземпляры классов (в том смысле, что графический интерфейс позволяет изменить значение поля `pay`, но не позволяет вызвать метод `giveRaise`).
- Если бы мы планировали распространять этот графический интерфейс, мы могли бы упаковать его в самостоятельную выполняемую программу – *скомпилированный двоичный файл* (frozen binary) – с использованием сторонних инструментов, таких как Py2Exe, PyInstaller и других (дополнительную информацию ищите в Интернете). Такие программы можно запускать, не устанавливая Python

на компьютер клиента, потому что интерпретатор байт-кода включается непосредственно в выполняемый файл.

Я оставлю все эти расширения для дальнейшего обдумывания и вернусь к некоторым из них далее в этой книге.

Два примечания, прежде чем двинуться дальше. Во-первых, я должен упомянуть, что программистам на языке Python доступно множество пакетов создания графических интерфейсов. Например, если вам потребуется реализовать графический интерфейс, состоящий не только из простых окон, вы сможете воспользоваться виджетом `Canvas` из библиотеки tkinter, поддерживающим возможность создания произвольной графики. Сторонние расширения, такие как Blender, OpenGL, VPython, PIL, VTK, Maya и PyGame, предоставляют еще более совершенные инструменты создания графических изображений, визуализации и воспроизведения анимационных эффектов для использования в сценариях на языке Python. Кроме того, возможности модуля `tkinter` могут быть расширены с помощью библиотек виджетов PMW, Tix и ttk, упоминавшихся ранее. Описание библиотек Tix и ttk вы найдете в руководстве по стандартной библиотеке Python, а также попробуйте поискать сторонние графические расширения на сайте PyPI или в Интернете.

Из уважения к поклонникам других инструментов создания графических интерфейсов, таких как wxPython и PyQt, я должен заметить, что существуют и другие средства разработки графических интерфейсов, выбор которых иногда зависит от личных предпочтений. Модуль `tkinter` был продемонстрирован здесь потому, что он является достаточно зрелым, надежным, полностью открытым, хорошо документированным, эффективно поддерживаемым и легковесным инструментом, входящим в состав стандартной библиотеки Python. По большому счету, он является стандартом для построения переносимых графических интерфейсов на языке Python.

Другие инструменты создания графических интерфейсов для сценариев на языке Python обладают своими достоинствами и недостатками, которые будут обсуждаться далее в книге. Например, за использование более богатых наборов виджетов приходится платить некоторым усложнением программного кода. Библиотека wxPython, к примеру, обладает более богатыми возможностями, но она значительно сложнее в использовании. Однако другие инструменты в значительной степени являются лишь вариациями на ту же тему – изучив один инструмент создания графического интерфейса, вы легко и просто овладеете и другими. Поэтому в этой книге мы не будем рассматривать примеры применения разных инструментов, а сосредоточимся на том, чтобы полностью освоить один инструмент.

Хотя программы с традиционными графическими интерфейсами, построенными, например, с помощью tkinter, при необходимости могут поддерживать доступ из сети, обычно они выполняются на единствен-

ном, отдельном компьютере. Иногда даже веб-страницы считаются разновидностью графического интерфейса, но, чтобы составить свое собственное мнение, вам необходимо прочитать следующий и последний раздел этой главы.

На досуге...

Конечно, библиотека обладает гораздо более широкими возможностями, чем было продемонстрировано в этом предварительном обзоре, и мы подробно будем знакомиться с ними далее в этой книге. В качестве еще одного небольшого примера, для демонстрации некоторых дополнительных возможностей библиотеки tkinter, ниже приводится сценарий *fungui.py*. В этом сценарии используется модуль `random` из библиотеки Python – для организации выбора из списка; конструктор `Toplevel` – для создания нового независимого окна; и функция обратного вызова `after` – для повторного вызова метода через указанное количество миллисекунд:

```
from tkinter import *
import random
fontsize = 30
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'cyan', 'purple']

def onSpam():
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color).pack(fill=BOTH)
    mainLabel.config(fg=color)

def onFlip():
    mainLabel.config(fg=random.choice(colors))
    main.after(250, onFlip)

def onGrow():
    global fontsize
    fontsize += 5
    mainLabel.config(font=('arial', fontsize, 'italic'))
    main.after(100, onGrow)

main = Tk()
mainLabel = Label(main, text='Fun Gui!', relief=RAISED)
mainLabel.config(font=('arial', fontsize, 'italic'),
                 fg='cyan',bg='navy')
mainLabel.pack(side=TOP, expand=YES, fill=BOTH)
Button(main, text='spam', command=onSpam).pack(fill=X)
Button(main, text='flip', command=onFlip).pack(fill=X)
Button(main, text='grow', command=onGrow).pack(fill=X)
main.mainloop()
```

Запустите этот сценарий, чтобы посмотреть, как он работает. Он создает главное окно с надписью внутри и тремя кнопками – щелчок на первой кнопке приводит к появлению нового окна с меткой, цвет которой выбирается случайным образом. Щелчок на двух других кнопках приводит к запуску независимых циклов вызовов методов обработчиков по таймеру, один из которых постоянно изменяет цвет надписи в главном окне, а другой постепенно увеличивает размер главного окна и шрифта надписи в нем. Но будьте внимательны, когда будете щелкать на последней кнопке, – изменение размеров выполняется со скоростью 10 раз в секунду, поэтому не упустите возможность закрыть окно, пока оно не убежало от вас. Эй, я же предупредил вас!

Шаг 6: добавляем веб-интерфейс

Графические интерфейсы проще в использовании, чем командная строка, и зачастую это все, что нам требуется, чтобы упростить доступ к данным. Однако, обеспечивая доступ к нашей базе данных из Интернета, мы открываем ее для более широкого круга пользователей. Любой, кто обладает выходом в Интернет и имеет броузер, сможет получить доступ к данным, независимо от того, где он находится и какой операционной системой пользуется. Годится любое устройство, от рабочей станции до сотового телефона. Кроме того, при наличии веб-интерфейса требуется только веб-броузер – чтобы получить доступ к данным, не нужно устанавливать Python, за исключением установки на сервере. Традиционные веб-интерфейсы обычно уступают в удобстве и скорости графическим интерфейсам, однако их переносимость может иметь решающее значение.

Как будет показано далее в этой книге, существуют различные способы реализации интерактивных веб-страниц, обеспечивающих доступ к нашим данным. Для решения простых задач, как наша, CGI-сценариев, выполняющихся на стороне сервера, будет более чем достаточно. Поскольку это, пожалуй, самый простой подход и к тому же являющийся основой для более совершенных технологий, разработка CGI-сценариев хорошо подходит для изучения основ разработки веб-приложений.

Для создания более сложных приложений существует богатое многообразие инструментальных средств и фреймворков для Python – включая Django, TurboGears, Google App Engine, pylons, web2py, Zope, Plone, Twisted, CherryPy, Webware, mod_python, PSP и Quixote, – упрощающих решение типичных задач и предоставляющих инструменты, которые в противном случае может потребоваться реализовать самостоятельно. Новейшие технологии, такие как Flex, Silverlight и pyjamas (версия фреймворка Google Web Toolkit, перенесенная на язык Python,

и компилятор с языка Python на язык JavaScript), предлагают дополнительные пути создания интерактивных и динамических пользовательских интерфейсов веб-страниц и открывают дверь к использованию Python в разработке полнофункциональных интернет-приложений (Rich Internet Applications, RIA).

Я еще вернусь к этим инструментам позднее, а пока не будем усложнять задачу и напишем CGI-сценарий.

Основы CGI

Писать CGI-сценарии на языке Python достаточно просто, если уже имеется опыт работы с формами HTML, адресами URL и имеется некоторое представление об особенностях модели клиент/сервер Интернета (все эти темы мы будем обсуждать далее в этой книге). Вы можете знать или не знать все подробности, но в основном модель взаимодействий вам должна быть знакома.

В двух словах: пользователь приходит на веб-сайт и получает форму HTML для заполнения в броузере. После отправки формы на сервере запускается сценарий, указанный либо в самой форме, либо в адресе URL сервера, который в ответ воспроизводит другую страницу HTML. В такой схеме взаимодействий данные обычно проходят через три программы: от броузера клиента они передаются веб-серверу, затем CGI-сценарию и возвращаются обратно броузеру. Это естественная модель взаимодействия с базами данных, которой мы будем следовать, – пользователь будет отправлять серверу ключ в базе данных и в ответ будет получать соответствующую страницу с записью.

Далее в книге мы подробнее познакомимся с основами CGI, а пока, в качестве первого примера, создадим простой интерактивный сценарий, который будет запрашивать имя пользователя и возвращать его обратно веб-броузеру. Первая страница в этом примере – это просто форма ввода, реализованная в виде разметки HTML, как показано в примере 1.30. Этот файл HTML хранится на веб-сервере и передается веб-броузеру, выполняющемуся на компьютере клиента.

Пример 1.30. PP4E\Preview\cgi101.html

```
<html>
<title>Interactive Page</title>
<body>
<form method=POST action="cgi-bin/cgi101.py">
    <P><B>Enter your name:</B>
    <P><input type=text name=user>
    <P><input type=submit>
</form>
</body></html>
```

Обратите внимание, что в атрибуте `action` этой формы определяется сценарий, который будет обрабатывать ее. Эта страница будет возвра-

щаться при обращении по ее адресу URL. После получения клиентом эта форма в окне броузера будет выглядеть, как показано на рис. 1.10 (в данном случае, в Internet Explorer).

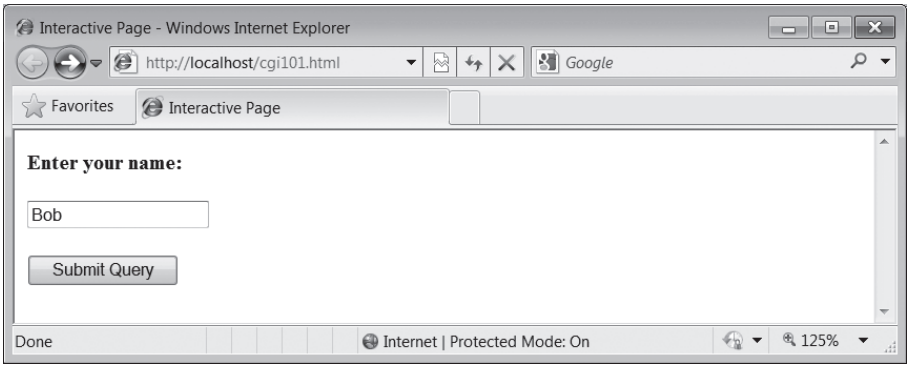


***Рис. 1.10.** Форма ввода на странице cgi101.html*

После отправки формы клиентом веб-сервер получит запрос (подробнее о веб-сервере чуть ниже) и запустит CGI-сценарий на языке Python, представленный в примере 1.31. Как и файл HTML, этот сценарий также находится на веб-сервере – он выполняется на стороне сервера, обрабатывает введенные данные и воспроизводит ответ, который отправляется броузеру на стороне клиента. Сценарий использует модуль `cgi`, чтобы извлечь данные из формы и вставить их в поток разметки HTML ответа с соответствующим экранированием. Модуль `cgi` обеспечивает интерфейс к полям ввода формы, отправленной броузером, напоминающий интерфейс словаря, и передачу разметки HTML, которую выводит сценарий, броузеру для отображения в виде следующей страницы. В мире CGI поток стандартного вывода соединен с клиентом посредством сокета.

Пример 1.31. PP4E\Preview\cgi-bin\cgi101.py

```
#!/usr/bin/python
import cgi
form = cgi.FieldStorage()              # парсинг данных формы
print('Content-type: text/html\n')     # http-заголовок плюс пустая строка
print('<title>Reply Page</title>')     # html-разметка ответа
if not 'user' in form:
    print('<h1>Who are you?</h1>')
else:
    print('<h1>Hello <i>%s</i>!</h1>' % cgi.escape(form['user'].value))
```

И если все пройдет как надо, мы получим в ответ страницу, изображенную на рис. 1.11, которая, по сути, просто выводит данные, введенные на странице с формой ввода. Страница на этом рисунке была воспроизведена разметкой HTML, которую вывел CGI-сценарий на стороне сер-

вера. В данном случае имя пользователя сначала было передано со стороны клиента на сервер, а затем обратно, возможно преодолев по пути многие сети и километры. Безусловно, это очень небольшой веб-сайт, но одни и те же принципы действуют для любого сайта, выводит он просто введенные данные или является полноценным сайтом электронного бизнеса.

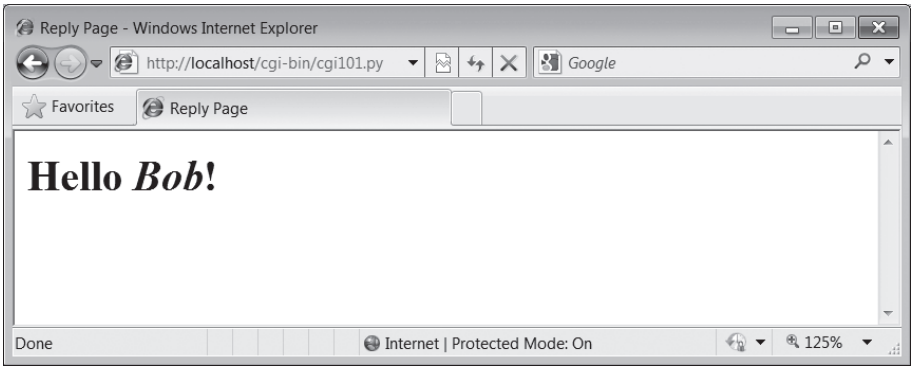


Рис. 1.11. *Страница ответа, воспроизведенная сценарием cgi101.py в ответ на получение формы ввода*

Если у вас возникли проблемы с организацией этих взаимодействий в Unix-подобной системе, попробуйте изменить путь к интерпретатору Python в строке `#!`, находящейся в начале сценария, и дать файлу сценария право на выполнение командой `chmod`, хотя во многом это зависит от вашего веб-сервера (подробнее о сервере мы поговорим чуть ниже).

Обратите также внимание, что CGI-сценарий в примере 1.31 выводит не полную разметку HTML: здесь отсутствуют теги `<html>` и `<body>`, которые можно увидеть в примере 1.30. Строго говоря, эти теги следовало бы вывести, но веб-броузеры спокойно воспринимают их отсутствие, да и цель этой книги состоит вовсе не в том, чтобы обучить вас формальному языку разметки HTML, – более подробную информацию об HTML ищите в других источниках.

Графические и веб-интерфейсы

Прежде чем двинуться дальше, имеет смысл потратить минуту, чтобы сравнить этот простой пример CGI-сценария с простым графическим интерфейсом, реализованным в примере 1.28 и изображенным на рис. 1.6. В данном случае сценарии выполняются на стороне сервера и генерируют разметку HTML, которая отображается веб-броузером. В реализации графического интерфейса мы вызываем функции, конструирующие интерфейс на экране и реагирующие на события, которые выполняются в рамках единого процесса и на одном и том же компьютере. Графический интерфейс состоит из нескольких программных

уровней, но целиком реализован в единственной программе. Реализация на основе CGI, напротив, имеет распределенную архитектуру – сервер, броузер и, возможно, сам CGI-сценарий выполняются как отдельные программы, обычно взаимодействующие друг с другом по сети.

Учитывая эти различия, автономная модель реализации графического интерфейса выглядит более простой и непосредственной: в ней отсутствует промежуточный сервер; чтобы получить ответ, не требуется вызывать новую программу; не требуется генерировать разметку HTML и в нашем распоряжении имеется вся мощь инструмента создания графического интерфейса. С другой стороны, веб-интерфейс может отображаться в любом броузере, на любом компьютере и для его работы необходимо установить Python только на сервере.

Еще больше ситуацию запутывает то обстоятельство, что графический интерфейс также может использовать сетевые инструменты из стандартной библиотеки Python для получения и отображения данных, хранящихся на удаленном сервере (то есть так же, как это делают броузеры). Некоторые новейшие фреймворки, такие как Flex, Silverlight и pyjamas, предоставляют инструменты реализации более полнофункциональных пользовательских интерфейсов в веб-страницах (полнофункциональных интернет-приложений, упоминавшихся выше), хотя и ценой сложности программного кода и большего количества программных уровней. Далее в книге мы еще вернемся к обсуждению различий между графическим интерфейсом и CGI, потому что на сегодняшний день это и есть основной выбор. А теперь рассмотрим несколько практических проблем, связанных с работой механизма CGI, прежде чем применить его к нашей базе данных.

Запуск веб-сервера

Для запуска CGI-сценариев нам потребуется веб-сервер, который будет обслуживать наши страницы HTML и запускать сценарии на языке Python по запросам. Сервер является необходимым промежуточным звеном между броузером и CGI-сценарием. Если у вас нет учетной записи на компьютере, где уже установлен такой веб-сервер, вам придется запустить собственный веб-сервер. Мы могли бы настроить полноценный веб-сервер промышленного уровня, такой как свободно распространяемый веб-сервер Apache (в котором, кстати, можно настроить поддержку Python с помощью расширения `mod_python`). Однако для данной главы я написал на языке Python собственный простой веб-сервер, программный код которого приводится в примере 1.32.

Мы еще вернемся к инструментам, использованным в этом примере, далее в книге. Тем не менее замечу, что в стандартной библиотеке Python уже имеется реализация некоторых типов сетевых серверов, благодаря чему мы можем реализовать CGI-совместимый и переносимый веб-сервер, написав всего 8 строк программного кода (точнее, 16, если учесть комментарии и пустые строки).

Далее в этой книге мы увидим, насколько просто создать собственный сетевой сервер, используя низкоуровневые функции для работы с сокетами в Python, однако в стандартной библиотеке уже имеются реализации многих наиболее распространенных типов серверов. Модуль `socketserver`, например, поддерживает многопоточные и ветвящиеся версии серверов TCP и UDP. Еще большее количество реализаций можно найти в сторонних системах, таких как Twisted. Модули из стандартной библиотеки, использованные в примере 1.32, предоставляют все, что необходимо для обслуживания нашего веб-содержимого.

Пример 1.32. PP4E\Preview\webserver.py

```
"""
Реализация веб-сервера на языке Python, способная запускать серверные
CGI-сценарии на языке Python; обслуживает файлы и сценарии в текущем
рабочем каталоге; сценарии на языке Python должны находиться в каталоге
webdir\cgi-bin или webdir\htbin;
"""
import os, sys
from http.server import HTTPServer, CGIHTTPRequestHandler

webdir = '.'   # место, где находятся файлы html и подкаталог cgi-bin
port = 80      # по умолчанию http://localhost/, иначе используйте
               # http://localhost:xxxx/
os.chdir(webdir)               # перейти в корневой каталог HTML
srvraddr = ("", port)          # имя хоста и номер порта
srvrobj = HTTPServer(srvraddr, CGIHTTPRequestHandler)
srvrobj.serve_forever()        # запустить как бесконечный фоновый процесс
```

Классы, используемые сценарием, предполагают, что обслуживаемые файлы HTML находятся в текущем рабочем каталоге, а запускаемые CGI-сценарии находятся в подкаталоге *cgi-bin* или *htbin*. Как следует из имени файла в примере 1.31, для сценариев мы будем использовать подкаталог *cgi-bin*. Некоторые веб-серверы определяют CGI-сценарии по расширению в именах файлов, однако мы будем считать CGI-сценариями все файлы, находящиеся в определенном каталоге.

Чтобы запустить веб-сервер, достаточно запустить этот сценарий (из командной строки, щелчком на ярлыке или иным способом). Он будет выполняться бесконечно, ожидая запросы от броузеров и других клиентов. Сервер ожидает запросы, направляемые на компьютер, где он выполняется, прослушивая стандартный порт HTTP с номером 80. Чтобы использовать этот сценарий для обслуживания других веб-сайтов, необходимо либо запустить его из другого каталога, содержащего файлы HTML и подкаталог *cgi-bin* со сценариями CGI, либо изменить значение переменной `webdir`, записав в нее имя корневого каталога сайта (сценарий автоматически перейдет в этот каталог и будет обслуживать файлы, находящиеся в нем).

Но где в киберпространстве фактически выполняется сценарий сервера? Если посмотреть внимательнее, на рисунках в предыдущем разделе

можно заметить, что в адресной строке броузера (в верхней части окна, сразу после последовательности символов *http://*) всегда используется имя сервера *localhost*. Чтобы не усложнять, я запустил веб-сервер на том же компьютере, где запускается веб-броузер, а это означает, что сервер будет иметь имя «localhost» (и соответствующий IP-адрес «127.0.0.1»). То есть клиент и сервер – это один и тот же компьютер: клиент (веб-броузер) и сервер (веб-сервер) – это просто разные процессы, одновременно выполняющиеся на одном и том же компьютере.

Хотя этот веб-сервер не может использоваться в промышленных целях, тем не менее он отлично подходит для тестирования CGI-сценариев – вы можете разрабатывать их на том же самом компьютере без необходимости перемещать программный код на удаленный сервер после каждого изменения. Просто запустите этот сценарий из каталога, где находятся файлы HTML и подкаталог *cgi-bin* с CGI-сценариями, и затем вводите в броузере адрес *http://localhost/...*, чтобы получить доступ к своим HTML-страницам и сценариям. Ниже приводится вывод, полученный от сценария веб-сервера в окне консоли в ОС Windows, который был запущен на том же компьютере, что и веб-броузер, из каталога, где находятся файлы HTML:

```
...\PP4E\Preview> python webserver.py
mark-VAIO - - [28/Jan/2010 18:34:01] "GET /cgi101.html HTTP/1.1" 200 -
mark-VAIO - - [28/Jan/2010 18:34:12] "POST /cgi-bin/cgi101.py HTTP/1.1" 200 -
mark-VAIO - - [28/Jan/2010 18:34:12] command: C:\Python31\python.exe -u C:\
Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\cgi-bin\cgi101.py ""
mark-VAIO - - [28/Jan/2010 18:34:13] CGI script exited OK
mark-VAIO - - [28/Jan/2010 18:35:25] "GET /cgi-bin/cgi101.py?user=Sue+Smith
HTTP/1.1" 200 -
mark-VAIO - - [28/Jan/2010 18:35:25] command: C:\Python31\python.exe -u C:\
Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\cgi-bin\cgi101.py
mark-VAIO - - [28/Jan/2010 18:35:26] CGI script exited OK
```

Здесь следует сделать одно замечание: на некоторых платформах, чтобы запустить сервер, прослушивающий порт по умолчанию с номером 80, вам могут потребоваться привилегии администратора, поэтому узнайте, как в вашей системе запустить такой сервер, или попробуйте использовать порт с другим номером. Чтобы задействовать порт с другим номером, измените значение переменной `port` в сценарии и указывайте его явно в адресной строке броузера (например, *http://localhost:8888/*). Подробнее об этом соглашении будет рассказываться далее в книге.

Чтобы запустить этот сервер на удаленном компьютере, выгрузите файлы HTML и подкаталог с CGI-сценариями на удаленный компьютер, запустите там этот сценарий, а в адресной строке броузера вместо имени «localhost» используйте доменное имя или IP-адрес удаленного компьютера (например, *http://www.myserver.com/*). При использовании удаленного сервера все взаимодействия будут протекать, как показано здесь, но при этом запросы и ответы будут передаваться не между при-

ложениями на одном и том же компьютере, а автоматически будут направляться через сетевые соединения.

Чтобы покопаться в реализации серверных классов нашего веб-сервера, обращайтесь к файлам с исходными текстами в стандартной библиотеке Python (*C:\Python31\Lib* для версии Python 3.1). Одно из основных преимуществ открытых систем, таких как Python, состоит в том, что мы всегда можем заглянуть «под капот». В главе 15 мы расширим пример 1.32 возможностью указывать имя каталога и номер порта из командной строки.

Использование строки запроса и модуля urllib

В простом CGI-сценарии, представленном выше, мы запускали сценарий на языке Python, заполняя и отправляя форму, содержащую имя сценария. На практике серверные CGI-сценарии могут вызываться разными способами – либо за счет отправки формы, как было показано выше, либо за счет отправки серверу строки URL (интернет-адреса), содержащей входные данные в конце. Такую строку URL можно отправить не только с помощью броузера, то есть минуя традиционный путь, лежащий через форму ввода.

Например, на рис. 1.12 изображена страница, сгенерированная в ответ на ввод адреса URL в адресной строке броузера (символ + здесь означает пробел):

```
http://localhost/cgi-bin/cgi101.py?user=Sue+Smith
```

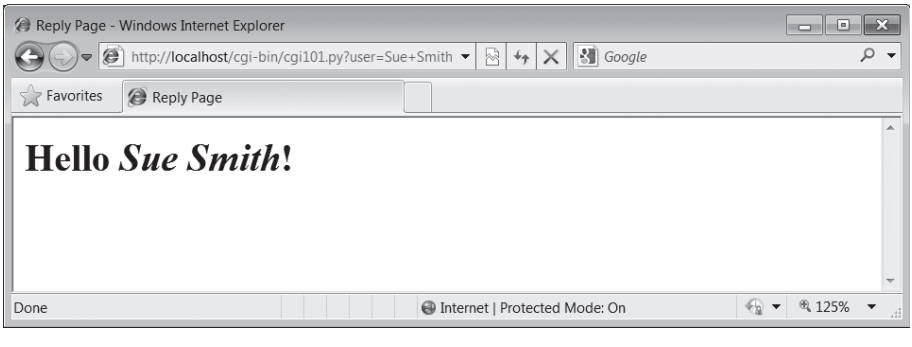


***Рис. 1.12.** Ответ сценария cgi101.py на запрос GET с параметрами*

Входные данные здесь, известные как *параметры запроса*, находятся в конце строки URL, после символа `?`. Они не были введены в поля формы. Строку URL с дополнительными входными данными иногда называют GET-запросом. Наша оригинальная форма отправляет запрос методом POST, в котором входные данные отправляются отдельно. К счастью, в CGI-сценариях на языке Python не требуется различать эти два вида запросов – парсер входных данных в модуле `cgi` автоматически обрабатывает все различия между методами отправки данных.

Вполне возможно и часто даже полезно иметь возможность отправлять входные данные в строке URL в виде параметров запроса вообще без помощи веб-броузера. Пакет `urllib` из стандартной библиотеки Python, например, позволяет читать ответ, сгенерированный сервером для любого допустимого адреса URL. Фактически он позволяет посещать веб-страницы или вызывать CGI-сценарии из другого сценария – ваш программный код на языке Python будет играть роль веб-клиента. Ниже демонстрируется пример использования пакета в интерактивной оболочке:

```
>>> from urllib.request import urlopen
>>> conn = urlopen('http://localhost/cgi-bin/cgi101.py?user=Sue+Smith')
>>> reply = conn.read()
>>> reply
b'<title>Reply Page</title>\n<h1>Hello <i>Sue Smith</i>!</h1>\n'

>>> urlopen('http://localhost/cgi-bin/cgi101.py').read()
b'<title>Reply Page</title>\n<h1>Who are you?</h1>\n'

>>> urlopen('http://localhost/cgi-bin/cgi101.py?user=Bob').read()
b'<title>Reply Page</title>\n<h1>Hello <i>Bob</i>!</h1>\n'
```

Пакет `urllib` реализует интерфейс получения ответов от сервера для заданной строки URL, напоминающий интерфейс файлов. Обратите внимание, что ответ, который мы получаем от сервера, представляет собой простую разметку HTML (обычно отображается броузером). Мы можем обрабатывать этот текст с помощью любых инструментов обработки текста, входящих в состав Python, включая:

- Строковые методы поиска и разбиения
- Модуль `re`, позволяющий выполнять сопоставление с регулярными выражениями
- Развитую поддержку парсинга разметки HTML и XML в стандартной библиотеке, включая модуль `html.parser`, а также SAX-, DOM- и ElementTree-подобные инструменты парсинга разметки XML.

В сочетании с такими инструментами пакет `urllib` естественным образом подходит для применения в интерактивном тестировании веб-сайтов, в собственных клиентских графических интерфейсах, в программах, извлекающих содержимое веб-страниц, и в автоматизированных регрессионных тестах для тестирования удаленных CGI-сценариев.

Форматирование текста ответа

Еще одно, последнее примечание: так как для взаимодействия с клиентами CGI-сценарии используют текст, они должны форматировать его, следуя определенному набору правил. Например, обратите внимание, что в примере 1.31 между заголовком ответа и разметкой HTML при-

сутствует пустая строка в виде явного символа перевода строки (`\n`), в дополнение к символу перевода строки, который автоматически выводится функцией `print`, – это обязательный разделитель.

Кроме того, обратите внимание, что текст, добавляемый в разметку HTML ответа, передается через вызов функции `cgi.escape` (она же `html.escape` в Python 3.2 – смотрите примечание в разделе «Инструменты экранирования HTML и URL в языке Python», в главе 15), на тот случай, если он содержит символы, имеющие специальное значение в HTML. Например, на рис. 1.13 изображена страница ответа, полученная в результате ввода имени пользователя `Bob </i> Smith`, – последовательность символов `</i>` в середине преобразуется этой функцией в последовательность `&lt;/i&gt;`, благодаря чему исключается влияние этой последовательности на фактическую разметку HTML (воспользуйтесь возможностью просмотра исходного кода страницы, имеющейся в броузерах, чтобы убедиться в этом). Без вызова этой функции остаток имени был бы выведен обычным, некурсивным шрифтом.

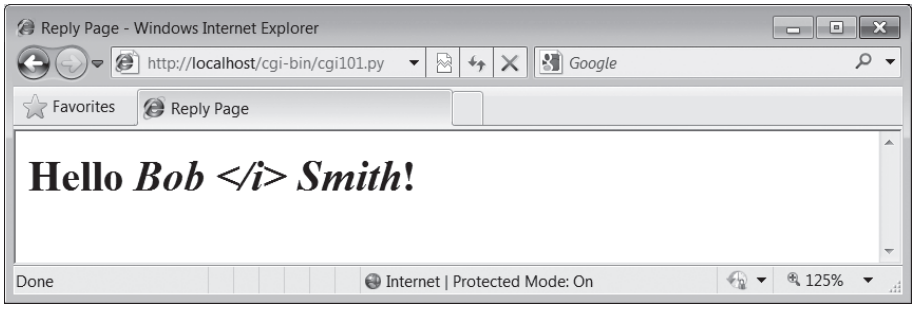


***Рис. 1.13.** Экранирование символов HTML*

Экранирование текста, как в данном примере, требуется не всегда, но его следует применять, когда содержимое текста заранее не известно, – сценарии, генерирующие разметку HTML, должны следовать правилам ее оформления. Как мы увидим далее в этой книге, похожая функция `urllib.parse.quote` применяет правила экранирования к тексту в строке с адресом URL. Кроме того, мы увидим, что крупные фреймворки часто решают задачи форматирования текста автоматически.

Веб-интерфейс к хранилищу с данными

Теперь для создания нашего приложения баз данных на основе технологии CGI, представленной в предыдущем разделе, нам потребуется реализовать более крупную форму ввода и отображения данных. На рис. 1.14 изображена форма, которую мы реализуем для доступа к нашей базе данных.

Реализация веб-сайта

Чтобы обеспечить возможность взаимодействий, создадим разметку HTML начальной формы ввода, а также CGI-сценарий на языке Python, который будет отображать полученные результаты и обрабатывать запросы на изменение данных в хранилище. В примере 1.33 приводится разметка HTML формы ввода, которая создает страницу, изображенную на рис. 1.14.

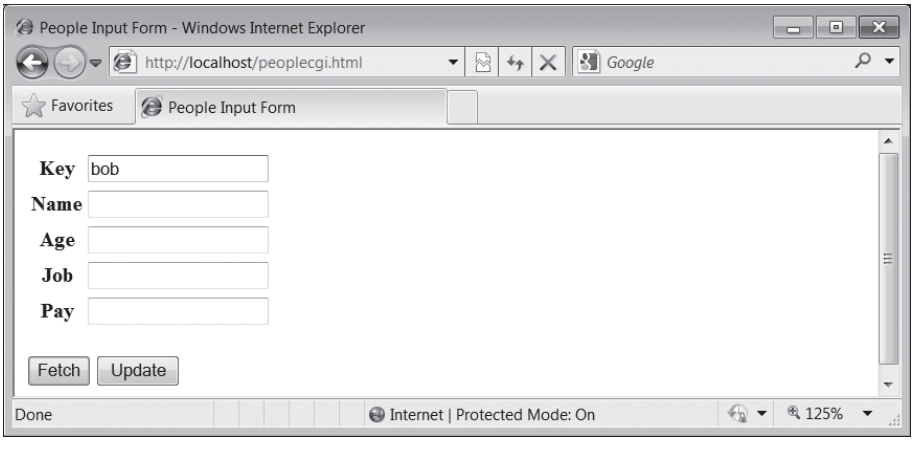


***Рис. 1.14.** Форма ввода peoplecgi.html*

Пример 1.33. PP4E\Preview\peoplecgi.html

```
<html>
<title>People Input Form</title>
<body>
<form method=POST action="cgi-bin/peoplecgi.py">
    <table>
    <tr><th>Key <td><input type=text name=key>
    <tr><th>Name<td><input type=text name=name>
    <tr><th>Age <td><input type=text name=age>
    <tr><th>Job <td><input type=text name=job>
    <tr><th>Pay <td><input type=text name=pay>
    </table>
    <p>
    <input type=submit value="Fetch", name=action>
    <input type=submit value="Update", name=action>
</form>
</body></html>
```

Обработкой формы и других запросов будет заниматься CGI-сценарий на языке Python, представленный в примере 1.34, который будет извлекать и изменять записи в нашем хранилище. Обратно он будет возвращать страницу, похожую на ту, что воспроизводит разметка в при-

мере 1.33, но с полями формы, заполненными значениями атрибутов объекта, извлеченного из хранилища.

Как и в реализации графического интерфейса, для вывода результатов и ввода изменений будет использоваться одна и та же веб-страница. В отличие от графического интерфейса, этот сценарий будет запускаться заново в ответ на каждое действие пользователя и каждый раз снова будет открывать базу данных (атрибут `action` формы содержит ссылку на сценарий для следующего запроса). Модель CGI не предоставляет возможности сохранения информации о состоянии между запросами, поэтому каждый раз мы вынуждены выполнять все действия с самого начала.

Пример 1.34. PP4E\Preview\cgi-bin\peoplecgi.py

```
"""
Реализует веб-интерфейс для просмотра и изменения экземпляров классов
в хранилище; хранилище находится на сервере (или на том же компьютере,
если используется имя localhost)
"""

import cgi, shelve, sys, os            # cgi.test() выведет поля ввода
shelvename = 'class-shelve'            # файлы хранилища находятся
                                       # в текущем каталоге
fieldnames = ('name', 'age', 'job', 'pay')

form = cgi.FieldStorage()              # парсинг данных формы
print('Content-type: text/html')       # заголовок + пустая строка для ответа
sys.path.insert(0, os.getcwd())        # благодаря этому модуль pickle
                                       # и сам сценарий будут способны
                                       # импортировать модуль person
# главный шаблон разметки html
replyhtml = """
<html>
<title>People Input Form</title>
<body>
<form method=POST action="peoplecgi.py">
    <table>
    <tr><th>key<td><input type=text name=key value="%(key)s">
    $ROWS$
    </table>
    <p>
    <input type=submit value="Fetch", name=action>
    <input type=submit value="Update", name=action>
</form>
</body></html>
"""

# вставить разметку html с данными в позицию $ROWS$
rowhtml = '<tr><th>%s<td><input type=text name=%s value="%%(%s)s">\n'
rowshtml = ''
```

```
for fieldname in fieldnames:
    rowshtml += (rowhtml % ((fieldname,) * 3))
replyhtml = replyhtml.replace('$ROWS$', rowshtml)

def htmlize(adict):
    new = adict.copy()                        # значения могут содержать &, >
    for field in fieldnames:                  # и другие специальные символы,
        value = new[field]                    # отображаемые особым образом;
        new[field] = cgi.escape(repr(value)) # их необходимо экранировать
    return new

def fetchRecord(db, form):
    try:
        key = form['key'].value
        record = db[key]
        fields = record.__dict__              # для заполнения строки ответа
        fields['key'] = key                   # использовать словарь атрибутов
    except:
        fields = dict.fromkeys(fieldnames, '?')
        fields['key'] = 'Missing or invalid key!'
    return fields

def updateRecord(db, form):
    if not 'key' in form:
        fields = dict.fromkeys(fieldnames, '?')
        fields['key'] = 'Missing key input!'
    else:
        key = form['key'].value
        if key in db:
            record = db[key]                  # изменить существующую запись
        else:
            from person import Person         # создать/сохранить новую
            record = Person(name='?', age='?') # eval: строки должны быть
                                              # заключены в кавычки
        for field in fieldnames:
            setattr(record, field, eval(form[field].value))
        db[key] = record
        fields = record.__dict__
        fields['key'] = key
    return fields

db = shelve.open(shelvename)
action = form['action'].value if 'action' in form else None
if action == 'Fetch':
    fields = fetchRecord(db, form)
elif action == 'Update':
    fields = updateRecord(db, form)
else:
    fields = dict.fromkeys(fieldnames, '?')      # недопустимое значение
    fields['key'] = 'Missing or invalid action!' # кнопки отправки формы
```

```
db.close()
print(replyhtml % htmlize(fields))              # заполнить форму ответа
                                                # из словаря
```

Сценарий получился таким большим, потому что в его задачу входит обработка ввода пользователя, выполнение операций с базой данных и генерирование разметки HTML для ответа. Тем не менее действует он достаточно прямолинейно и по своему поведению напоминает реализацию графического интерфейса из предыдущего раздела.

Каталоги, форматирование строк и безопасность

Прежде чем двинуться дальше, необходимо сделать несколько важных замечаний. Прежде всего, сценарий веб-сервера, представленный в примере 1.32, должен быть запущен до того, как вы начнете экспериментировать, – он будет перехватывать запросы от броузера и передавать их нашему CGI-сценарию.

Обратите также внимание, что при запуске CGI-сценарий добавляет путь к текущему рабочему каталогу (`os.getcwd`) в путь поиска модулей `sys.path`. Не изменяя переменную окружения `PYTHONPATH`, этот прием позволит модулю `pickle` и самому сценарию импортировать модуль `person`, находящийся в том же каталоге, что и сценарий. Из-за нового способа запуска CGI-сценариев, реализованного в Python 3, текущий рабочий каталог не добавляется в список `sys.path` автоматически, хотя при этом файлы хранилища, находящиеся там, будут обнаруживаться и открываться корректно. Эта особенность в поведении может отличаться, в зависимости от выбранного веб-сервера.

Еще один интересный прием в CGI-сценарии – использование словаря атрибутов записи (`__dict__`) как источника значений в операции экранирования полей внутри выражения форматирования строки, преобразующего строку шаблона HTML в ответ, в последней строке сценария. Напомню, что выражение вида `%(key)code` заменит ключ `key` значением этого ключа в словаре:

```
>>> D = {'say': 5, 'get': 'shrubbery'}
>>> D['say']
5
>>> S = '%(say)s => %(get)s' % D
>>> S
'5 => shrubbery'
```

Благодаря использованию словаря атрибутов мы можем ссылаться на атрибуты по их именам в форме строк. Фактически часть шаблона ответа генерируется программным кодом. Если его структура кажется вам непонятной, просто вставьте инструкции вывода `replyhtml` и вызова `sys.exit` и запустите сценарий из командной строки. Ниже показано, как выглядит разметка HTML таблицы в середине сгенерированного ответа (немного отформатированная здесь для удобочитаемости):

```
<table>
<tr><th>key<td><input type=text name=key value="%(key)s">
<tr><th>name<td><input type=text name=name value="%(name)s">
<tr><th>age<td><input type=text name=age value="%(age)s">
<tr><th>job<td><input type=text name=job value="%(job)s">
<tr><th>pay<td><input type=text name=pay value="%(pay)s">
</table>
```

Далее этот текст заполняется значениями ключей из словаря атрибутов записи инструкцией форматирования строки в конце сценария. Эта инструкция выполняется после того, как словарь будет обработан вспомогательной функцией, преобразующей значения в текст с помощью функции `repr` и экранирующей текст вызовом функции `cgi.escape`, в соответствии с требованиями языка разметки HTML (опять же, последний шаг не всегда является обязательным, но он никогда не будет лишним).

Эти строки ответа в формате HTML можно было бы жестко определить в программном коде, но генерирование их из кортежа с именами полей обеспечивает более универсальное решение – в будущем мы сможем добавлять новые поля без необходимости изменять шаблон HTML. Инструменты обработки строк в языке Python позволяют это.

Справедливости ради следует заметить, что более новый метод `str.format` позволяет добиться того же эффекта, что и традиционный оператор `%` форматирования, используемый в сценарии, и дает возможность использовать синтаксис ссылок на атрибуты объектов, который выглядит более явным по сравнению с приемом использования ключей словаря `__dict__`:

```
>>> D = {'say': 5, 'get': 'shrubbery'}

>>> '%(say)s => %(get)s' % D           # выражение: ссылка на ключ
'5 => shrubbery'
>>> '{say} => {get}'.format(**D)       # метод: ссылка на ключ
'5 => shrubbery'

>>> from person import Person
>>> bob = Person('Bob', 35)

>>> '%(name)s, %(age)s' % bob.__dict__ # выражение: ключи __dict__
'Bob, 35'
>>> '{0.name} => {0.age}'.format(bob)  # метод: синтаксис атрибутов
'Bob => 35'
```

Однако из-за того, что нам сначала необходимо экранировать атрибуты, мы не можем использовать синтаксис атрибутов в вызове метода форматирования. Фактически для выбора нам доступен лишь синтаксис ссылок на ключи, представленный выше. (К моменту написания этих строк еще не было очевидно, какой из двух способов форматирования займет доминирующее положение, поэтому мы позволим себе исполь-

зовать в книге оба способа – даже если какой-то из этих способов заменит другой, вы все равно останетесь в выигрыше.)

В интересах безопасности необходимо также напомнить, что прием использования функции `eval` для преобразования входных данных в объекты языка Python является достаточно мощным, но далеко не безопасным. Эта функция с радостью выполнит любой программный код на языке Python, который в свою очередь сможет выполнить любые системные операции, разрешение на которые будет иметь процесс сценария. Если проблема безопасности имеет для вас значение, то вам придется обеспечить выполнение сценария в ограниченном окружении или использовать более специализированные механизмы преобразования, такие как функции `int` и `float`. Вообще говоря, проблема безопасности занимает важное место в мире веб-приложений, где строки запросов могут поступать из самых разных источников. Однако, поскольку все мы здесь считаемся друзьями, мы проигнорируем возможную угрозу.

Пользование веб-сайтом

Несмотря на сложности, связанные с серверами, каталогами и строками, пользоваться веб-интерфейсом ничуть не сложнее, чем графическим интерфейсом. Вдобавок веб-интерфейс имеет дополнительное преимущество – им можно пользоваться в любой операционной системе, где имеется броузер и подключение к Интернету. Чтобы извлечь запись из хранилища, заполните поле Key (Ключ) и щелкните на кнопке Fetch (Извлечь) – сценарий заполнит страницу данными, полученными из атрибутов соответствующего экземпляра класса, извлеченного из хранилища, как показано на рис. 1.15, где была извлечена запись с ключом `bob`.

На рис. 1.15 показано, что получится, когда ключ передается с помощью формы. Как уже отмечалось выше, CGI-сценарий можно также

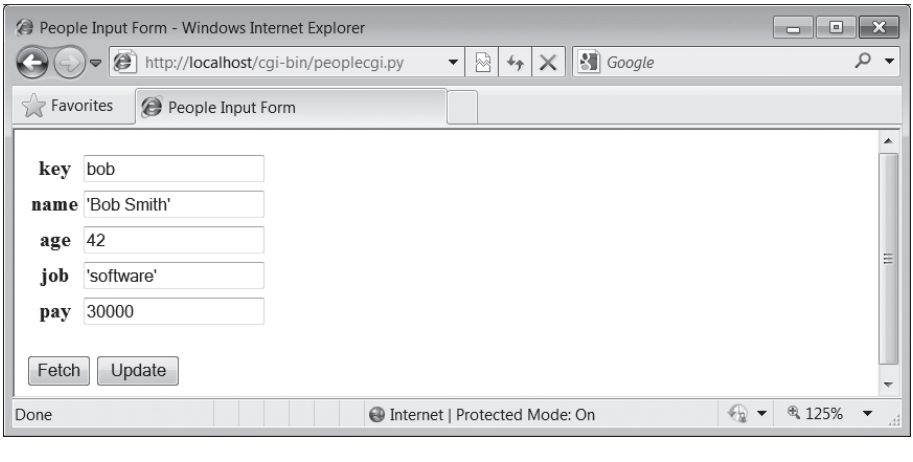


***Рис. 1.15.** Страница ответа peoplecgi.py*

вызвать, передав входные данные в виде строки запроса, поместив ее в конец адреса URL. На рис. 1.16 показана страница, полученная в ответ на попытку обратиться по следующему адресу URL:

```
http://localhost/cgi-bin/peoplecgi.py?action=Fetch&key=sue
```

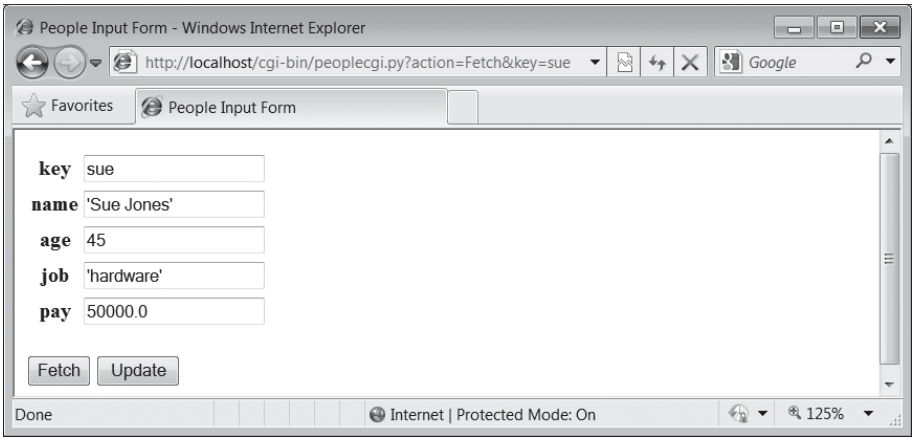


***Рис. 1.16.** Ответ сценария peoplecgi.py на запрос с параметрами*

Как вы уже знаете, такую строку URL можно отправить с помощью броузера или сценария, использующего такие инструменты, как пакет `urllib`. И снова, замените «localhost» на доменное имя своего сервера, если вы запускаете сценарий на удаленном компьютере.

Чтобы изменить запись, извлеките ее по ключу, введите новые значения в поля ввода и щелкните на кнопке `Update` (Изменить) – сценарий извлечет значения из полей ввода и запишет их в соответствующие атрибуты экземпляра класса в хранилище. На рис. 1.17 показана страница ответа, полученная после изменения записи с ключом `sue`.

Наконец, операция добавления новой записи выполняется точно так же, как и в графическом интерфейсе: укажите новые значения ключа и полей, щелкните на кнопке `Update` (Изменить) – CGI-сценарий создаст новый экземпляр класса, запишет в его атрибуты значения соответствующих полей ввода и сохранит его в хранилище с новым ключом. В действительности здесь под покровом веб-страницы выполняются операции с объектом класса, но нам не приходится иметь дело с логикой его создания. На рис. 1.18 изображена запись, добавленная в базу данных таким способом.

В принципе мы точно так же можем изменять и добавлять записи, отправляя соответствующие строки URL – из броузера или из сценария – например:

```
http://localhost/cgi-bin/
    peoplecgi.py?action=Update&key=sue&pay=50000&name=Sue+Smith& ...и далее...
```

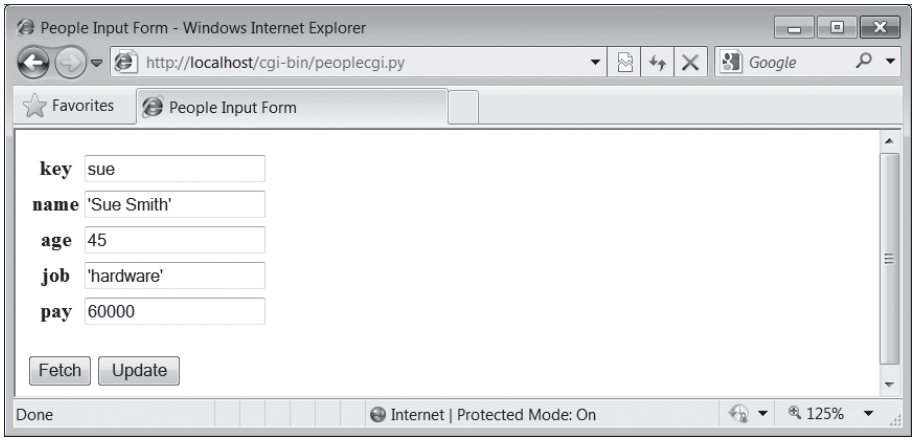


Рис. 1.17. *Ответ peoplecgi.py на операцию изменения записи*

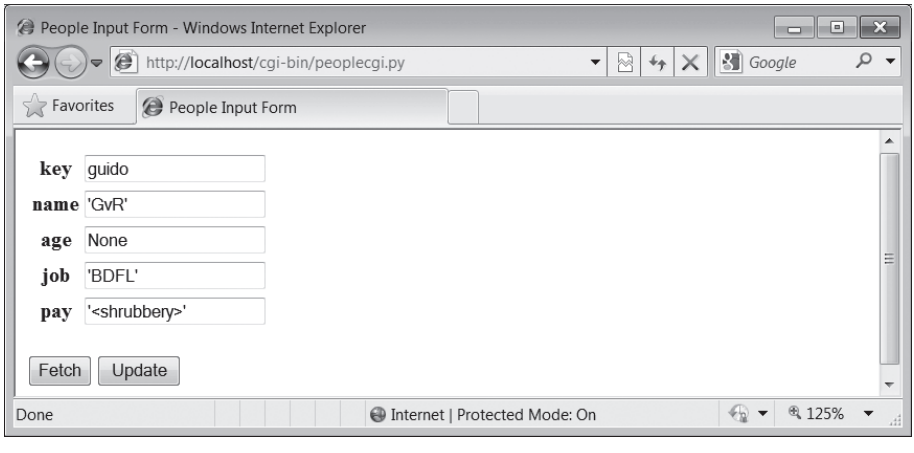


Рис. 1.18. *Ответ peoplecgi.py после добавления новой записи*

Однако вводить такую длинную строку URL без использования автоматизированных инструментов существенно сложнее, чем заполнять поля формы. Ниже приводится часть страницы ответа, сгенерированной в ответ на создание записи с ключом «guido» и изображенной на рис. 1.18 (воспользуйтесь возможностью просмотра исходного кода страницы, имеющейся в броузерах, чтобы убедиться в этом). Обратите внимание, что символы < и > были преобразованы функцией `cgi.escape` в экранированные последовательности HTML, перед тем как они были вставлены в ответ:

```
<tr><th>key<td><input type=text name=key value="guido">
<tr><th>name<td><input type=text name=name value="'GvR'">
<tr><th>age<td><input type=text name=age value="None">
```

```
<tr><th>job<td><input type=text name=job value="'BDFL'">
<tr><th>pay<td><input type=text name=pay value="'&lt;shrubbery&gt;'">
```

Как обычно, для тестирования нашего CGI-сценария можно использовать пакет `urllib` из стандартной библиотеки – возвращаемый результат представляет собой простую разметку HTML, которую можно проанализировать с помощью других инструментов, имеющихся в стандартной библиотеке, и использовать в качестве основы для системы регрессионного тестирования серверного сценария, выполняющейся на любой машине, подключенной к Интернету. Мы могли бы даже реализовать анализ ответа сервера, полученного таким способом, и отображать данные в графическом интерфейсе, реализованном с помощью библиотеки tkinter, – графические интерфейсы и веб-страницы не являются взаимоисключающими технологиями. В последнем примере получения данных в интерактивном сеансе демонстрируется фрагмент страницы HTML с сообщением об ошибке, которая была сгенерирована в ответ на отсутствующее или недопустимое входное значение, с разрывами строк, добавленными для удобочитаемости:

```
>>> from urllib.request import urlopen
>>> url = 'http://localhost/cgi-bin/peoplecgi.py?action=Fetch&key=sue'
>>> urlopen(url).read()
b'<html>\n<title>People Input Form</title>\n<body>\n
<form method=POST action="peoplecgi.py">\n <table>\n
<tr><th>key<td><input type=text name=key value="sue">\n
<tr><th>name<td><input type=text name=name value="\'Sue Smith\'">\n
<tr><t ...остальной текст удален...

>>> urlopen('http://localhost/cgi-bin/peoplecgi.py').read()
b'<html>\n<title>People Input Form</title>\n<body>\n
<form method=POST action="peoplecgi.py">\n <table>\n
<tr><th>key<td><input type=text name=key value="Missing or invalid action!">\n
    <tr><th>name<td><input type=text name=name value="\'?\'">\n
<tr><th>age<td><input type=text name=age value="\'?\'">\n<tr> ...остальной текст
удален...
```

Фактически, если CGI-сценарий выполняется на локальном компьютере «localhost», для просмотра одного и того же хранилища вы сможете использовать и графический интерфейс из предыдущего раздела, и веб-интерфейс из этого раздела – это всего лишь альтернативные интерфейсы доступа к одним и тем же хранимым объектам Python. Для сравнения на рис. 1.19 показано, как выглядит запись в графическом интерфейсе, которую мы видели на рис. 1.18, – это тот же самый объект, но на этот раз мы получили ее, не обращаясь к промежуточному серверу, запускающему другие сценарии или генерирующему разметку HTML.

Как и прежде, мы всегда можем проверить результаты нашей деятельности на сервере, используя интерактивную оболочку или другие сценарии. Мы можем просматривать содержимое базы данных с помощью

веб-броузеров или графического интерфейса, но в любом случае это всего лишь объекты Python в файле хранилища:

```
>>> import shelve
>>> db = shelve.open('class-shelve')
>>> db['sue'].name
'Sue Smith'
>>> db['guido'].job
'BDFL'
>>> list(db['guido'].name)
['G', 'v', 'R']
>>> list(db.keys())
['sue', 'bill', 'nobody', 'tomtom', 'tom', 'bob', 'peg', 'guido']
```

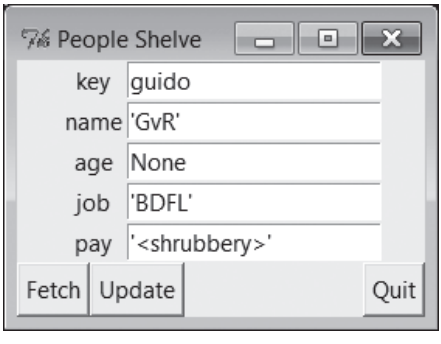


Рис. 1.19. *Тот же самый объект, отображаемый в графическом интерфейсе*

Ниже приводятся результаты запуска первоначального сценария из примера 1.19, извлекающего информацию из базы данных, который мы написали до того, как перешли к реализации графического и веб-интерфейса, – в языке Python существует масса способов просмотра данных:

```
...\PP4E\Preview> dump_db_classes.py
sue =>
   Sue Smith 60000
bill =>
   bill 9999
nobody =>
   John Doh None
tomtom =>
   Tom Tom 40000
tom =>
   Tom Doe 90000
bob =>
   Bob Smith 30000
peg =>
   1 4
```

```
guido =>
   GvR <shrubbery>
Smith
Doe
```

Дальнейшие направления усовершенствования

Естественно, что в этот пример можно было бы внести множество улучшений:

- Разметка HTML для начальной формы ввода, представленная в примере 1.33, несколько избыточна для сценария в примере 1.34, и ее можно было бы генерировать с помощью другого сценария, используемого как источник совместно используемой информации.
- Фактически мы вообще можем отказаться от встраивания разметки HTML в наш сценарий, если воспользуемся одним из инструментов-генераторов HTML, с которыми мы познакомимся далее в книге, таким как HTMLgen (система создания разметки HTML из дерева объектов документа) и PSP (Python Server Pages – серверные страницы Python, серверная система шаблонов HTML для Python, напоминающая PHP и ASP).
- Чтобы упростить обслуживание, можно было бы также вынести разметку HTML для CGI-сценария в отдельный файл, чтобы отделить представление от логики (с разными файлами могли бы работать разные специалисты).
- Кроме того, если веб-сайтом могут пользоваться сразу несколько человек, мы могли бы добавить возможность блокировки файла хранилища или перейти на использование *базы данных*, такой как ZODB или MySQL, чтобы обеспечить возможность параллельных изменений. ZODB и другие полноценные системы управления базами данных позволяют также использовать возможность отмены транзакций в случае ошибок. Реализовать простейшую блокировку файла можно с помощью функции `os.open` и ее флагов.
- Механизмы *ORM* (object relational mappers – объектно-реляционного отображения) для Python, такие как SQLObject и SQLAlchemy, упоминавшиеся выше, также способны обеспечить поддержку одновременной работы нескольких пользователей с реляционной базой данных, сохраняя в ней представление данных в виде наших классов Python.
- Наконец, если размер нашего сайта станет больше, чем несколько интерактивных страниц, мы могли бы перейти от CGI-сценариев к более развитым веб-фреймворкам, таким как упоминавшиеся в начале этого раздела – Django, TurboGears, pyjamas и другие. На случай, если потребуется сохранять информацию между обращениями к страницам, можно было бы использовать такие инструменты, как cookies, скрытые поля ввода, сеансы, поддерживаемые модулем mod_python и FastCGI.

- Если потребуется хранить на сайте информационное наполнение, производимое его пользователями, мы могли бы перейти на использование Plone. Это популярная и открытая система управления содержимым, написанная на языке Python, использующая сервер приложений Zope, реализующая модель документооборота и делегирующая управление содержимым сайта его авторам.
- А если на повестке дня встанет поддержка *беспроводных* или *распределенных* интерфейсов, мы могли бы перенести нашу систему на сотовые телефоны, используя один из трансляторов с языка Python, доступных, например, для платформы Nokia и Google Android, или на платформу распределенных вычислений, такую как Google App Engine. Язык Python с успехом проникает в области, куда ведет развитие технологий.

Но, тем не менее, и графический, и веб-интерфейс, созданные нами, вполне справляются со своей работой.

Конец демонстрационного примера

На этом мы заканчиваем знакомство с вводным демонстрационным примером использования языка Python. Мы исследовали способы представления данных, ООП, механизмы сохранения объектов, инструменты создания графических интерфейсов и основы разработки веб-сайтов. Ни одну из этих тем мы не рассматривали достаточно глубоко. Тем не менее хотелось бы надеяться, что эта глава пробудила в вас любопытство к программированию приложений на языке Python.

В оставшейся части книги мы углубимся в изучение этих и других инструментов и тем прикладного программирования, чтобы помочь вам включить язык Python в работу в ваших собственных программах. В следующей главе мы начнем наше путешествие с изучения инструментов системного программирования и администрирования, имеющихся в распоряжении программистов на языке Python.

Скрытые «сюрпризы» в Python

На настоящий момент, когда я пишу эти строки в 2010 году, я занимаюсь языком Python уже почти 18 лет, и я видел, как он вырос из никому не известного языка в инструмент, который в том или ином виде используется практически каждой организацией, занимающейся разработкой, и входит в четверку или пятерку наиболее используемых языков программирования в мире. Это были лучшие годы.

Оглядываясь назад, могу сказать, что если в языке Python что-то и осталось действительно неизменным, так это его врожденная

способность заставлять разработчиков акцентировать внимание на качестве программного кода. И это практически неизбежно. Язык, который требует, чтобы разработчик форматировал программный код для большей удобочитаемости, не может не заставить людей поднимать вопросы о выборе наиболее удачных приемов разработки программного обеспечения.

Пожалуй, ничто так не подчеркивает эту сторону жизни языка Python, как модуль `this` из стандартной библиотеки – своего рода сюрприз, или «пасхальное яйцо» в Python, созданный одним из основных разработчиков Python – Тимом Петерсом (Tim Peters), который хранит в себе список основных принципов, на которых основывается язык. Чтобы увидеть их, запустите интерактивный сеанс интерпретатора Python и импортируйте модуль (естественно, он доступен на всех платформах):

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
(Перевод:
Дзен языка Python, составлен Тимом Петерсом

Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Плоское лучше, чем вложенное.
Разреженное лучше, чем плотное.
```

```
Удобочитаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
При этом практичность важнее безупречности.
Ошибки никогда не должны замалчиваться.
Если не замалчиваются явно.
Встретив двусмысленность, отбрось искушение угадать.
Должен существовать один и, желательно, только один очевидный способ
сделать что-то.
Хотя он поначалу может быть и не очевиден, если вы не голландец.
Сейчас лучше, чем никогда.
Хотя никогда зачастую лучше, чем *прямо сейчас*.
Если реализацию сложно объяснить – идея плоха.
Если реализацию легко объяснить – идея, возможно, хороша.
Пространства имен – отличная штука! Будем делать их побольше!
)[1]
```

Особого упоминания заслуживает правило «Явное лучше, чем неявное», которое в мире Python известно, как аббревиатура «EIBTI» («Explicit is better than implicit») – одна из основных идей языка Python, и одно из самых сильных отличий от других языков. Любой, кто проработал на этой ниве более, чем несколько лет, сможет засвидетельствовать, что волшебство и инженерное искусство есть вещи несовместимые. Конечно, сам язык Python не всегда неукоснительно следовал всем этим правилам, но старался придерживаться их как можно ближе. И если Python заставляет людей задумываться о таких вещах, то это уже победа. Кстати, название языка отлично смотрится на футболке.

[1] Перевод взят из Википедии: *http://ru.wikipedia.org/wiki/Python – Прим. перев.*

II

Системное программирование

В этой первой посвященной деталям части книги представлены инструменты Python для системного программирования – интерфейсы к службам операционной системы, а также к контексту выполнения программы. Эта часть книги состоит из следующих глав:

Глава 2

Эта глава обеспечивает полный обзор часто используемых инструментов системного программирования. Она неторопливо знакомит с инструментами и приемами, которые мы будем использовать далее в этой книге, и отчасти может использоваться, как справочник.

Глава 3

Эта глава продолжает тему, начатую в главе 2, и показывает, как работать со стандартными потоками ввода-вывода, аргументами командной строки, переменными окружения и многим другим.

Глава 4

Эта глава продолжает знакомить нас с системными интерфейсами и описывает инструменты и приемы, используемые при работе с файлами и каталогами. В этой главе мы познакомимся с двоичными файлами, с приемами обхода деревьев и так далее.

Глава 5

Эта глава служит введением в поддержку библиотекой Python параллельного выполнения программ. Здесь вы найдете описание потоков выполнения, механизма ветвления процессов, каналов, сокетов, сигналов, очередей и тому подобного.

Глава 6

Последняя глава этой части содержит коллекцию типичных примеров системного программирования, основанных на материале первых четырех глав. Сценарии Python, представленные здесь, выполняют реальные задачи; демонстрируют среди прочего, как разрезать и объединять файлы, сравнивать и копировать каталоги, тестировать другие программы, а также отыскивать и запускать выполняемые файлы.

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

2

Системные инструменты

«os.path – дорога к знанию»

В этой главе начинается детальное рассмотрение способов применения Python для решения практических задач программирования. В этой и в последующих главах мы увидим, как использовать Python для разработки системных инструментов, графических интерфейсов пользователя, приложений баз данных, веб-сценариев, веб-сайтов и многого другого. Попутно на практических примерах будут также изучаться важнейшие концепции программирования на Python: повторное использование программного кода, простота сопровождения, объектно-ориентированное программирование (ООП) и так далее.

В этой части книги мы начнем свое путешествие по программированию на языке Python с исследования *области системных приложений* – сценариев, работающих с файлами, программами и окружением программ в целом. Хотя примеры из этой области сфокусированы на определенном типе задач, используемые в них приемы окажутся полезными и в последующих частях книги. Иными словами, если вы еще не эксперт по системному программированию на Python, вам следует пускаться в путь именно с этого места.

Зачем здесь нужен Python?

Системные интерфейсы Python обслуживают области приложений, но в последующих пяти главах большинство примеров будет относиться к категории *системных инструментов* – программ, иногда называемых утилитами командной строки, сценариями командной оболочки, программами системного администрирования, системными программами и другими сочетаниями этих слов. Независимо от того, знакомы

ли вам эти 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 неочевидных 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 обширное многообразие встроенных библиотек упрощают (и даже делают приятным) использование развитых системных инструментов, таких как потоки выполнения, сигналы, ветвление процессов, сокеты и аналогичные им; такие инструменты намного сложнее использовать в неясном синтаксисе языков оболочек и в многоэтапных циклах разработки на компилируемых языках. Поддержка в Python таких идей, как ясность программного кода и объектно-ориентированное программирование, способствует созданию таких инструментов оболочки, которые можно читать, сопровождать и повторно 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 разработки системных инструментов, но он также обеспечивает *переносимость* сценариев. При использовании стандартной библиотеки Python большинство системных сценариев, написанных на языке Python, автоматически становятся переносимыми на все основные платформы. Например, сценарий для обработки каталогов, написанный под Windows, обычно может выполняться и под Linux безо всякой правки исходных текстов: достаточно просто скопировать сценарий. Для разработки сценариев, обеспечивающих такой уровень переносимости, необходимо прикладывать некоторые усилия, тем не менее при разумном использовании Python может стать единственным средством, которым необходимо владеть для создания системных сценариев.

В следующих пяти главах

Чтобы упростить изучение этой части книги, я поделил ее на пять глав:

- В этой главе я познакомлю вас с основными системными модулями в виде краткого обзора. Здесь мы впервые встретим некоторые из наиболее часто используемых системных инструментов.
- В главе 3 мы продолжим исследование основных системных интерфейсов – изучением их роли в терминах системного программирования, таких как потоки ввода-вывода, аргументы командной строки, переменные окружения и так далее.
- В главе 4 мы сосредоточимся на изучении инструментов Python для работы с файлами, каталогами и деревьями каталогов.

- В главе 5 мы перейдем к изучению стандартных инструментов Python для реализации параллельной обработки данных – процессов, потоков выполнения, очередей, каналов, сигналов и многих других.
- В главе 6, завершающей эту часть книги, будет представлена коллекция законченных системных программ. Здесь приводятся более крупные примеры, имеющие практическую ценность; в них для решения практических задач используются инструменты, представленные в четырех предыдущих главах. В состав этой коллекции входят не только универсальные системные сценарии, но и сценарии для обработки каталогов с файлами.

В примерах, представленных в последней главе этой части книги, мы будем уделять большое внимание не только системным интерфейсам, но и принципам разработки программ на языке Python в целом. Попутно будут представлены процедурные и объектно-ориентированные версии одних и тех же примеров, чтобы показать преимущества стратегического мышления.

«Батарейки – в комплекте»

В данной главе и в следующих за ней речь идет одновременно о языке Python и о его *стандартной библиотеке* – коллекции модулей, написанных на языке Python и C, которые автоматически устанавливаются вместе с интерпретатором. Хотя Python и представляет собой легкий язык сценариев, большая часть операций в реальных разработках на Python выполняется с привлечением этой обширной библиотеки инструментов (по последним подсчетам – несколько сотен модулей), поставляемых вместе с пакетом Python.

В действительности стандартная библиотека обладает настолько широкими возможностями, 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 практической повседневной деятельности уже присутствует в стандартной библиотеке и может быть импортировано. Несмотря на то, что стандартная библиотека не является частью самого языка, тем не менее она является стандартной частью системы Python, и можно быть уверенным, что она будет доступна везде, где выполняются сценарии. Фактически это одно из наиболее существенных отличий Python от некоторых других языков сценариев – благодаря тому, что в составе Python поставляется огромное количество библиотечных инструментов, для программистов на Python вспомогательные сайты не имеют такого большого значения, как CPAN для программистов на Perl.

Как будет показано далее, стандартные библиотеки выполняют существенную часть задачи программирования на Python. Овладев основами языка, вы заметите, что в основном заняты применением встроенных функций и модулей, поставляемых вместе с системой. С другой стороны, библиотеки обеспечивают все самое интересное. В реальном мире программы становятся интересными, когда в них начинают использоваться службы, находящиеся вне интерпретатора языка: сети, файлы, графические интерфейсы, базы данных и так далее. Все это поддерживается стандартной библиотекой Python.

Помимо стандартной библиотеки для Python существуют *дополнительные пакеты*, созданные сторонними разработчиками, которые могут быть получены и установлены отдельно. Когда писалась эта книга, большинство таких расширений сторонних разработчиков можно было найти путем поиска в Интернете и по ссылкам на *http://www.python.org* и на веб-сайте PyPI (ссылка на который также приводится на сайте *http://www.python.org*). Некоторые сторонние расширения являются крупными системами. Например, расширения NumPy, Django и VPython реализуют операции векторной алгебры, обеспечивают конструирование сайтов и предоставляют средства визуализации соответственно.

Если нужно сделать с помощью Python что-либо специальное, есть большая вероятность, что необходимая поддержка уже включена в состав стандартной библиотеки или реализована в виде бесплатного модуля с открытым исходным кодом. Большинство инструментов, используемых в этой книге, входит в стандартный дистрибутив Python, а то, что должно устанавливаться отдельно, будет отмечено особо. Конечно, идиома повторного использования программного кода делает программы зависимыми от используемого в них кода, однако на практике, в чем мы еще не раз убедимся в этой книге, мощные библиотеки с открытыми исходными текстами развиваются очень быстро, не запирая своих пользователей в узком кругу фиксированных возможностей и ограничений.

Знакомство с разработкой системных сценариев

Исследование области системного программирования мы начнем с краткого обзора модулей `sys` и `os` из стандартной библиотеки, а затем перейдем к более важным понятиям системного программирования. Из перечня атрибутов этих модулей можно заключить, что это очень крупные модули, – следующий пример интерактивного сеанса был получен в Python 3.1 и в Windows 7 вне среды IDLE:

```
C:\...\PP4E\System> python
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (...)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys, os
>>> len(dir(sys))          # 65 атрибутов
65
>>> len(dir(os))           # в Windows 122 атрибута, в Unix - больше
122
>>> len(dir(os.path))      # модуль, вложенный в os
52
```

Содержимое этих двух модулей может отличаться для разных версий Python и платформ. Например, модуль `os` имеет намного больший размер после сборки Python 3.1 из исходных текстов под Cygwin (Cygwin – система, обеспечивающая Unix-подобную функциональность в Windows; о ней рассказывается во врезке «Подробнее о Cygwin Python для Windows» в главе 5):

```
$ ./python.exe
Python 3.1.1 (r311:74480, Feb 20 2010, 10:16:52)
[GCC 3.4.4 (cygming special, gdc 0.12, using dmd 0.125)] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys, os
>>> len(dir(sys))
64
>>> len(dir(os))
217
>>> len(dir(os.path))
51
```

Я не собираюсь представлять все элементы в каждом встроенном модуле, поэтому прежде всего я хочу показать, как самостоятельно получать более подробную информацию. Такая задача может служить формальным оправданием тому, что здесь будут представлены некоторые базовые понятия системного программирования. Попутно мы напишем код первого сценария для форматирования документации.

Системные модули Python

Большинство интерфейсов Python системного уровня находится всего в двух модулях: `sys` и `os`. Впрочем, это несколько упрощенное представление – к данной области относятся и другие стандартные модули. В их числе:

`glob`

Реализует механизм подстановки имен файлов

`socket`

Обеспечивает возможности создания сетевых соединений и взаимодействий между процессами (Inter-Process Communication, IPC)

`threading`, `_thread`, `queue`

Средства запуска и синхронизации параллельных потоков выполнения

`time`, `timeit`

Обеспечивают возможность получения информации о системном времени

`subprocess`, `multiprocessing`

Средства запуска и управления параллельными процессами

`signal`, `select`, `shutil`, `tempfile` *и другие*

Для решения других системных задач

Некоторые сторонние расширения, такие как pySerial (интерфейс к последовательному порту), Pexpect (механизм управления взаимодействиями между программами, напоминающий утилиту Expect) и даже Twisted (сетевой фреймворк), также могут быть отнесены к разряду системных инструментов. Кроме того, некоторые встроенные функции также в действительности являются системными интерфейсами – функция `open`, например, обеспечивает интерфейс к файловой системе. Но в общем и целом ядро арсенала системных инструментов Python образуют модули `sys` и `os`.

В теории, по крайней мере, модуль `sys` экспортирует компоненты, относящиеся к самому *интерпретатору* Python (например, путь поиска модулей), а модуль `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 модуле `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 на различных платформах, но для сценариев на языке 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 заключить из предыдущих абзацев, обучение системному программированию на языке Python в основном сводится к изучению

[1] Они могут проникать и в ваше подсознание. Новички Python иногда описывают такое явление, как «сны на Python» (попробуйте дать упрощенную интерпретацию по Фрейду…).

системных модулей Python. К счастью, существует множество источников информации, облегчающих эту задачу, – от атрибутов модуля до печатных справочников и книг.

Например, если потребуется узнать, какие элементы экспортирует встроенный модуль, можно прочесть соответствующий раздел руководства по библиотеке, исследовать его исходный код (все-таки Python является открытым программным обеспечением) или получить список атрибутов и строку документации в интерактивном режиме. Давайте импортируем модуль sys в Python 3.1 и посмотрим, что в нем находится:

```
C:\...\PP4E\System> python
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_
frames', '_getframe', 'api_version', 'argv', 'builtin_module_names',
'byteorder','call_tracing', 'callstats', 'copyright', 'displayhook',
'dllhandle', 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_
prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
'getcheckinterval', 'getdefaultencoding', 'getfilesystemencoding',
'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'gettrace',
'getwindowsversion', 'hexversion', 'int_info', 'intern', 'maxsize',
'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_
cache', 'platform', 'prefix', 'ps1', 'ps2',
'setcheckinterval', 'setfilesystemencoding', 'setprofile', 'setrecursionlimit',
'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_
info', 'warnoptions', 'winver']
```

Функция dir просто возвращает список строк с именами всех атрибутов для любого объекта, имеющего атрибуты; это удобная подсказка по содержимому модуля при работе в интерактивном режиме. Мы можем понять, например, что существует нечто с именем sys.version, поскольку имя version присутствует в списке, возвращаемом функцией dir. Если этого недостаточно, всегда можно обратиться к строке __doc__ встроенного модуля:

```
>>> sys.__doc__
"This module provides access to some objects used or maintained by the\
ninterpreter and to functions that interact strongly with the interpreter.\n\
nDynamic objects:\n\nargv -- command line arguments; argv[0] is the script
pathname if known \npath -- module search path; path[0] is the script directory,
else ''\nmodules -- dictionary of loaded modules\n\ndisplayhook -- called to
show results in an i
...далее следует еще много текста..."
```

Постраничный вывод строк документации

Встроенный атрибут __doc__ обычно содержит строку документации, которая может выглядеть несколько странно при отображении в таком виде, – это одна длинная строка с символами перевода строки, выво-

дящимися как \n, а не красивый список строк. Чтобы отформатировать эти строки и придать им более удобочитаемый вид, можно воспользоваться функцией print:

```
>>> print(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

...далее следует еще много текста...
```

Встроенная функция print, в отличие от механизма вывода в интерактивной оболочке, корректно интерпретирует символы перевода строки. К сожалению, функция print не реализует возможность прокрутки или постраничного просмотра и поэтому может быть неудобной на некоторых платформах. Большего успеха можно добиться с помощью других инструментов, таких как встроенная функция help:

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys

DESCRIPTION
    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

...далее следует еще много текста...
```

Функция help – это один из интерфейсов, предоставляемых системой PyDoc. Она входит в состав стандартной библиотеки, распространяемой вместе с Python, и предназначена для отображения в форматированном виде документации (строк документации, а также дополнительной

структурной информации), связанной с объектом. Документация может быть в формате страниц справочного руководства Unix, который используется для вывода с помощью функции help, или в виде HTML-страницы, что еще лучше. Это очень удобный способ получения начальной информации при работе в интерактивном режиме, и это последний шанс разобраться, прежде чем погрузиться в справочники и книги.

Сценарий постраничного вывода

Функция help, с которой мы только что познакомились, также не обладает достаточной гибкостью при отображении информации. Хотя она и пытается в некоторых ситуациях обеспечить постраничный вывод, тем не менее на некоторых компьютерах – из тех, на которых мне приходилось работать, – она неточно выбирает размер страницы. Кроме того, она вообще не обеспечивает постраничный просмотр в графическом интерфейсе IDLE; вместо этого предлагается использовать полосу прокрутки, что весьма неудобно на больших мониторах. Когда мне требуется получить более полный контроль над тем, как функция help будет выводить текст, я обычно использую свой собственный вспомогательный сценарий, представленный в примере 2.1.

Пример 2.1. PP4E\System\more.py

```
"""
разбивает строку или текстовый файл на страницы для интерактивного просмотра
"""
def more(text, numlines=15):
    lines = text.splitlines()          # подобно split('\n') но без '' в конце
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print(line)
        if lines and input('More?') not in ['y', 'Y']: break

if __name__ == '__main__':
    import sys                         # если запускается как сценарий
    more(open(sys.argv[1]).read(), 10) # отобразить постранично содержимое
                                       # файла, указанного в командной строке
```

Главной в этом файле является функция more, и если вы обладаете достаточными знаниями языка Python, чтобы читать эту книгу, вы без труда поймете ее. Она просто разбивает строку по символам перевода строки, а затем извлекается срез и выводится сразу несколько строк (по умолчанию 15), чтобы избежать прокрутки экрана. Выражение извлечения среза lines[:15] вернет первые 15 элементов списка, а выражение lines[15:] – последние. Чтобы изменить размер страницы, передайте требуемое число строк в аргументе numlines (например, в последней строке примера 2.1 в аргументе numlines функции more передается число 10).

Вызов строкового метода splitlines, используемый в этом сценарии, возвращает список подстрок, полученный в результате разбиения исходной строки по символам перевода строки (например, ["line", "line",...]). Альтернативный метод split позволяет получить похожий результат, но в последнем элементе массива он возвращает пустую строку, если исходная строка заканчивается символом \n:

```
>>> line = 'aaa\nbbb\nccc\n'

>>> line.split('\n')
['aaa', 'bbb', 'ccc', '']

>>> line.splitlines()
['aaa', 'bbb', 'ccc']
```

Как будет показано далее в главе 4, символом конца строки в сценариях на языке Python всегда является \n (обозначающий байт с числовым значением 10), вне зависимости от платформы. (Если вы еще не знаете, почему это имеет значение, – символы DOS \r отбрасываются при чтении.)

Основы использования строковых методов

Пример 2.1 – это простая программа на языке Python, но в ней представлены три важные темы, заслуживающие краткого упоминания: она использует строковые методы, осуществляет чтение из файла и предусматривает возможность импортирования или выполнения как самостоятельного сценария. Строковые методы в языке Python как таковые не являются системными инструментами, но их можно встретить в большинстве программ на языке Python. В действительности они будут постоянно встречаться на протяжении этой и последующих глав, поэтому коротко рассмотрим наиболее полезные инструменты, имеющиеся в наборе. В число строковых методов входят вызовы для поиска и замены:

```
>>> mystr = 'xxxSPAMxxx'
>>> mystr.find('SPAM')             # вернет индекс первого вхождения
3
>>> mystr = 'xxaaxxaa'
>>> mystr.replace('aa', 'SPAM')    # замена всех вхождений
'xxSPAMxxSPAM'
```

Вызов метода find возвращает смещение первого вхождения подстроки, а метод replace осуществляет глобальный поиск и замену. Как и все строковые операции, метод replace возвращает новую строку, оставляя исходную строку неизменной (напомню, что строки являются неизменяемыми объектами). Для всех этих методов подстроки являются просто строками; в главе 19 будет представлен модуль re, который позволяет использовать *шаблоны* регулярных выражений при поиске и замене.

В самых последних версиях Python имеется возможность использовать оператор `in` определения принадлежности, как альтернативу методу `find`, когда необходимо всего лишь получить ответ «да» или «нет» (он проверяет присутствие подстроки в строке). Существует также несколько методов, удаляющих пробельные символы из концов строки, что особенно полезно при работе с текстовыми строками, извлекаемыми из файла:

```
>>> mystr = 'xxxSPAMxxx'
>>> 'SPAM' in mystr            # проверка присутствия подстроки в строке
True
>>> 'Ni' in mystr              # если подстрока отсутствует
False
>>> mystr.find('Ni')
-1

>>> mystr = '\t Ni\n'
>>> mystr.strip()              # удалит пробельные символы
'Ni'
>>> mystr.rstrip()             # то же самое, но только с правого конца
'\t Ni'
```

Среди строковых методов имеются также функции, которые могут быть полезны, например, как инструменты преобразования регистра символов, а кроме того, в стандартной библиотеке имеется модуль `string`, определяющий некоторые полезные константы:

```
>>> mystr = 'SHRUBBERY'
>>> mystr.lower()              # преобразует регистр символов
'shrubbery'

>>> mystr.isalpha()            # проверяет содержимое
True
>>> mystr.isdigit()
False

>>> import string              # константы, например, для использования в 'in'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'

>>> string.whitespace          # пробельные символы
' \t\n\r\x0b\x0c'
```

Существуют также методы, позволяющие разбивать строки по подстрокам-разделителям и объединять их вместе, вставляя между ними подстроку. Эти средства будут изучены далее в этой книге, но в качестве знакомства покажем, как они работают:

```
>>> mystr = 'aaa,bbb,ccc'
>>> mystr.split(',')           # разбить в список подстрок
['aaa', 'bbb', 'ccc']
```

```
>>> mystr = 'a b\nc\nd'
>>> mystr.split()              # разделитель по умолчанию: пробельные символы
['a', 'b', 'c', 'd']

>>> delim = 'NI'
>>> delim.join(['aaa', 'bbb', 'ccc'])  # объединить подстроки из списка
'aaaNIbbbNIccc'

>>> ' '.join(['A', 'dead', 'parrot'])  # добавить пробел между подстроками
'A dead parrot'

>>> chars = list('Lorreta')    # преобразовать в список символов
>>> chars
['L', 'o', 'r', 'r', 'e', 't', 'a']
>>> chars.append('!')
>>> ''.join(chars)             # преобразовать в строку: с пустым разделителем
'Lorreta!'
```

Эти вызовы оказываются удивительно мощными. Например, строку с колонками данных, разделенными символами табуляции, можно разобрать по колонкам единственным вызовом метода `split`; сценарий *more.py*, представленный выше, использует разновидность `splitlines` этого метода, чтобы разбить строку в список строк. На практике вызов метода `replace` можно эмулировать с помощью комбинации `split`/`join`:

```
>>> mystr = 'xxaaxxaa'
>>> 'SPAM'.join(mystr.split('aa')) # усложненная версия str.replace!
'xxSPAMxxSPAM'
```

Запомните на будущее, что язык Python не предусматривает автоматического преобразования строк в числа и обратно, поэтому если в этом возникнет необходимость, такие преобразования необходимо выполнять явно:

```
>>> int("42"), eval("42")      # преобразование строки в целое число
(42, 42)

>>> str(42), repr(42)          # преобразование целого числа в строку
('42', '42')

>>> ("%d" % 42), '{:d}'.format(42) # с помощью оператора и метода форматиров.
('42', '42')

>>> "42" + str(1), int("42") + 1   # в операциях конкатенации и сложения
('421', 43)
```

В последней приведенной инструкции первое выражение выполняет конкатенацию строк (так как оба операнда являются строками), а второе выполняет сложение целых чисел (поскольку оба объекта являются числами). Python не делает предположений о том, какое преобразование вы могли иметь в виду, и не выполняет преобразования автоматически. Одно из главных правил интерпретатора Python – где только воз-

можно, избегать закулисных магических действий и попыток что-то угадывать. Более подробно о средствах для работы со строками будет рассказано далее (им посвящена целая глава в пятой части), но, кроме того, стоит посмотреть описание дополнительных строковых инструментов в руководстве по библиотеке.

Другие особенности строк в Python 3.X: Юникод и тип bytes

Строго говоря, история со строками в Python 3.X гораздо богаче, чем можно было бы заключить из вышесказанного. До сих пор было продемонстрировано, что объекты типа `str` являются последовательностями символов (точнее – «кодовыми пунктами» Юникода, представляющими «элементы» Юникода), которые могут быть не только символами ASCII, но и многобайтовыми символами Юникода и предусматривают возможность кодирования и декодирования вручную или автоматически при выполнении операций с текстовыми файлами. Строки в программном коде заключаются в кавычки (например, `'abc'`) и допускают использование дополнительного синтаксиса для представления символов, не входящих в набор ASCII (например, `'\xc4\xe8'`, `'\u00c4\u00e8'`).

Однако на самом деле в Python 3.X имеется два дополнительных строковых типа, поддерживающих большинство операций, которыми обладает тип `str`: тип `bytes` – последовательность коротких целых чисел для представления 8-битовых двоичных данных и тип `bytearray` – изменяемый вариант типа `bytes`. Как вы уже знаете, присутствие символа «b» перед открывающей кавычкой (например, `b'abc'`, `b'\xc4\xe8'`) говорит о том, что вы имеете дело с объектом типа `bytes`. Как будет показано в главе 4, файлы в Python 3.X также проявляют подобную двойственность: при работе в текстовом режиме используется тип `str` (при этом предусматриваются преобразования символов конца строки и символов Юникода, в соответствии с указанной кодировкой), а при работе в двоичном режиме используется тип `bytes` (в этом случае данные при чтении/записи не подвергаются преобразованиям). В главе 5 мы увидим такое же деление при работе с такими инструментами, как сокеты, которые на сегодняшний день работают со строками байтов

Текст Юникода используется в интернационализированных приложениях, и многие инструменты языка Python, ранее ориентированные на работу с двоичными данными, в настоящее время работают со строками байтов. К ним относятся некоторые инструменты для работы с файлами, которые мы встретим далее, такие как функция `open`, а также инструменты `os.listdir` и `os.walk`, которые мы будем изучать в последующих главах. Как будет показано ниже, даже простые инструменты для работы с каталогами должны иметь возможность обрабатывать символы Юникода в содержимом и в именах файлов. Кроме того, инструменты для сериализации объектов и анализа двоичных данных на сегодняшний день ориентированы на работу со строками байтов.

Далее в этой книге мы также увидим, что в настоящее время Юникод используется для представления текста в графических интерфейсах; для обмена данными по сети, в виде последовательностей байтов; в стандартных инструментах Интернета, таких как электронная почта; и даже в некоторых механизмах сохранения объектов, таких как файлы DBM и модуль `shelve`. Любой интерфейс, предусматривающий работу с текстом, на сегодняшний день обязательно предусматривает работу с Юникодом, потому что тип `str` представляет строки символов Юникода, а не только ASCII. Как только мы в этой книге доберемся до сферы программирования приложений, для большинства программистов на Python 3.X тема Юникода перестанет быть необязательной.

Мы отложим дальнейшее обсуждение Юникода, пока нам не представится возможность увидеть его в прикладном контексте и в практических программах. Более фундаментальное освещение поддержки текстовых и двоичных данных Юникода в строках и файлах вы найдете в четвертом издании книги «Изучаем Python». Эта книга официально посвящена основам языка, что предполагает углубленное рассмотрение тем и позволило отвести этой теме отдельную главу, занимающую 45 страниц.

Основы операций с файлами

Помимо обработки строк, сценарий *more.py* также использует файлы – он открывает 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 функцией `open`, являются составной частью самого базового языка Python, я могу предположить, что вы хотя бы бегло знакомы с ними. Но в случае, если эта глава попалась вам в самом начале изучения Python, подскажу, что следующие вызовы 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:

```
open('file').read()           # читает весь файл в строку
open('file').read(N)          # читает следующие N байтов в строку
open('file').readlines()      # читает весь файл в массив строк
open('file').readline()       # читает следующую строку, до символа '\n'
```

Как мы скоро увидим, эти вызовы можно также применять в Python к командам оболочки, чтобы прочитать их вывод. У объектов файлов есть также методы `write`, которые посылают строки в соответствующий файл. Более глубоко темы, связанные с файлами, раскрываются в главе 4, однако сами операции вывода данных в файл и чтения их обратно в языке Python реализуются очень просто:

```
>>> file = open('spam.txt', 'w')       # создать файл spam.txt
>>> file.write(('spam' * 5) + '\n')    # записать текст: вернет
21                                     # число записанных символов
```

```
>>> file.close()

>>> file = open('spam.txt')              # или open('spam.txt').read()
>>> text = file.read()                   # прочитать в строку
>>> text
'spamspamspamspamspam\n'
```

Два способа использования программ

Последние несколько строк в сценарии *more.py* из примера 2.1 знакомят нас с одним из первых важных понятий в программировании инструментов командной строки. Они настраивают файл так, чтобы его можно было использовать двумя способами: как *сценарий* или как *библиотеку*.

Напомню, что в каждом модуле Python доступна встроенная переменная `__name__`, в которую интерпретатор Python записывает значение `__main__`, только если файл выполняется как программа, а не импортируется в качестве библиотеки. Благодаря этому функция `more` в этом файле автоматически выполняется в последней строке файла, когда сценарий запускается, как самостоятельная программа, а не импортируется в какое-либо другое место. Этот простой прием является ключом к созданию многократно используемых сценариев: благодаря реализации логики программы в виде *функции*, а не в виде программного кода верхнего уровня, ее можно импортировать и повторно использовать в других сценариях.

В результате появляется возможность запускать *more.py* отдельно или импортировать его и вызывать функцию `more` из любого другого места. При запуске файла как самостоятельной программы мы передаем ей в командной строке имя файла, который нужно прочесть и выводить постранично: в следующей главе будет полностью описано, как слова, вводимые в команде для запуска программы, появляются во встроенном списке `sys.argv`. Ниже приводится пример запуска файла сценария для постраничного вывода самого себя (эта команда должна выполняться в каталоге *PP4E\System*, иначе входной файл не будет найден; причина этого будет пояснена позднее):

```
C:\...\PP4E\System> python more.py more.py
"""
разбивает строку или текстовый файл на страницы для интерактивного просмотра
"""
def more(text, numlines=15):
    lines = text.splitlines()          # подобно split('\n'), но без '' в конце
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print(line)
More?y
        if lines and input('More?') not in ['y', 'Y']: break
```

```
if __name__ == '__main__':
    import sys                          # если запускается как сценарий
    more(open(sys.argv[1]).read(), 10) # отобразить постранично содержимое
                                        # файла, указанного в командной строке
```

Если мы импортируем файл more.py, мы явно передаем строку в его функцию more; функция more – как раз такая утилита, которая нам нужна для просмотра текста документации. Запуск этой утилиты для просмотра строки документации модуля sys представит информацию о том, какие возможности дает этот модуль сценариям, в виде, пригодном для чтения:

```
C:\...\PP4E\System> python
>>> from more import more
>>> import sys
>>> 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

displayhook -- called to show results in an interactive session
excepthook -- called to handle any uncaught exception other than SystemExit
  To customize printing in an interactive session or to install a custom
  top-level exception handler, assign other functions to replace these.

stdin -- standard input file object; used by input()
More?
```

Нажатие клавиши y или Y заставит функцию отобразить несколько следующих строк документации и снова вывести приглашение, если список строк еще не закончился. Попробуйте сделать это у себя, и вы увидите, как выглядит оставшаяся часть строки документации. Кроме того, попробуйте поэкспериментировать, задавая размер окна во втором аргументе, – вызов more(sys.__doc__, 5) будет выводить текст блоками по 5 строк.

Руководства по библиотекам Python

Если изложение кажется недостаточно детальным, то полную информацию вы можете получить, обратившись к разделу, посвященному модулю sys, в руководстве по библиотекам Python. Все стандартные руководства Python доступны в Интернете и, кроме того, часто устанавливаются вместе с Python. В Windows стандартные руководства устанавливаются автоматически. Для обращения к руководствам приведу несколько простых указаний:

- В Windows щелкните на кнопке Пуск (Start), выберите пункт Все программы (All Programs), затем выберите пункт Python и пункт Python Manuals (Руководства Python). Руководства чудесным образом появятся на вашем экране. Начиная с версии Python 2.4 руководства для Windows поставляются в формате файлов справки, благодаря чему они поддерживают возможность поиска и навигации.
- В Linux или Mac OS X можно щелкнуть на элементе руководства в менеджере файлов или запустить броузер из командной строки и перейти в каталог, где в вашей системе находятся файлы HTML руководства.
- Если в вашей системе руководств не обнаружилось, их всегда можно прочесть в Интернете. Перейдите на веб-сайт Python *http://www.python.org* и найдите ссылки, ведущие к документации. Этот сайт также обеспечивает возможность простого поиска поруководствам.

В любом случае выберите руководство «Library», если вас интересуют такие вещи, как модуль `sys`. Это руководство содержит описание всех стандартных модулей, встроенных типов данных и функций и многое другое. В комплект стандартных руководств Python входит 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 сайте *http://www.python.org*. На сегодняшний день есть также коммерческие печатные справочники по Python, в том числе «Python Essential Reference»[1], «Python in a Nutshell», «Python Standard Library» и «Python Pocket Reference». Некоторые из этих книг являются более полными и содержат примеры, при этом последний из перечисленных справочников удобно использовать как «напоминалку», после того как вы уже раз-другой изучили библиотеку.[2]

---

[1] Д. Бизли «Python. Подробный справочник», СПб.: Символ-Плюс, 2010.

[2] Я написал последний справочник в качестве замены справочному приложению, имевшемуся в первом издании этой книги. Он задуман как дополнение к книге, которую вы читаете, а его последнее издание также может служить переправой для читателей, использующих версию Python 2.X. Как уже говорилось в предисловии, книга, которую вы сейчас держите, является учебником, а не справочником, поэтому вам, вероятно, придется в конечном счете найти какой-нибудь источник справочной информации (однако я настолько самонадеян, что предлагаю выбрать мой справочник).

Модуль sys

Но достаточно разговоров об источниках информации (и основах разработки сценариев) – перейдем к подробностям, касающихся системных модулей. Как говорилось выше, модули sys и os образуют ядро набора инструментов Python для решения системных задач. Сделаем сейчас краткий интерактивный обзор некоторых инструментов, имеющихся в этих двух модулях, прежде чем использовать их в более крупных примерах. Начнем с модуля sys, меньшего из этих двух модулей. Напомню, чтобы получить полный список всех атрибутов модуля sys, вы можете передать его функции dir (или посмотреть на список, полученный нами выше в этой главе).

Платформы и версии

Как и в большинстве модулей, в модуле sys есть атрибуты, содержащие информацию, и функции, выполняющие действия. Например, в его атрибутах можно найти название операционной системы, в которой выполняется программный код, наибольшее целое число, поддерживаемое аппаратной платформой на данном компьютере (хотя в Python 3.X целые числа могут быть произвольной величины), и номер версии интерпретатора Python, выполняющего программный код:

```
C:\...\PP4E\System> python
>>> import sys
>>> sys.platform, sys.maxsize, sys.version
('win32', 2147483647, '3.1.1 (r311:74483, Aug 17 2009, 17:02:12)
...дополнительные строки были удалены...')

>>> if sys.platform[:3] == 'win': print('hello windows')
...
hello windows
```

Если программный код должен по-разному выполняться на разных компьютерах, просто проверьте строку sys.platform, как сделано в этом примере. Несмотря на то, что Python по большей части независим от платформы, а непереносимые средства обычно заключаются в условные инструкции if, типа той, что здесь приведена, тем не менее далее, например, будет показано, что средства запуска программ и низкоуровневые взаимодействия с консолью различаются в зависимости от платформы. Поэтому проверка значения sys.platform поможет выбрать правильный инструмент для той машины, на которой выполняется сценарий.

Путь поиска модулей

Модуль sys позволяет также проверить путь поиска модулей как в интерактивном режиме, так и из программы на языке Python. Переменная sys.path хранит список строк, представляющих действительный путь

поиска в выполняющемся интерпретаторе Python. Когда выполняется операция импортирования модуля, Python просматривает этот путь слева направо, пытаясь отыскать файл модуля в каждом каталоге, указанном в списке. Поэтому данная переменная позволит вам убедиться, 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 и содержимого файлов *.pth*, находящихся в каталогах со сценариями. В действительности, если заглянуть в список sys.path в интерактивной оболочке, можно обнаружить довольно много каталогов, которые отсутствуют в переменной PYTHONPATH: в него входит также указатель на домашний каталог сценария (пустая строка – назначение которой я объясню далее, после знакомства с функцией os.getcwd) и набор каталогов стандартных библиотек, который может быть различным в каждой установке:

```
>>> sys.path
['', 'C:\\PP4thEd\\Examples', ...плюс каталоги стандартной библиотеки... ]
```

Как это ни удивительно, но список sys.path можно *изменять* программным способом. Сценарии могут использовать такие операции над списками, как append, extend, insert, pop и remove, а также использовать инструкцию del, чтобы изменять путь поиска модулей в процессе выполнения, чтобы подключить все каталоги с необходимыми модулями. Python всегда использует для импорта текущее значение sys.path, учитывая все внесенные вами изменения:

```
>>> sys.path.append(r'C:\mydir')
>>> sys.path
['', 'C:\\PP4thEd\\Examples', ...more deleted..., 'C:\\mydir']
```

Такое непосредственное изменение переменной sys.path является альтернативой установке переменной оболочки PYTHONPATH, хотя и не самой лучшей. Изменения в sys.path сохраняются лишь до завершения процесса Python, и их нужно повторно вносить при каждом новом запуске программы или сеанса Python. Однако некоторые типы программ (например, сценарии, выполняющиеся на веб-сервере) не должны зависеть от значения PYTHONPATH. Такие сценарии могут сами настраивать список sys.path при запуске и включать в него все необходимые каталоги с импортируемыми модулями. Более конкретный пример использования sys.path приводится в примере 1.34, в предыдущей главе, где мы вы-

[1] Может случиться, что Python видит переменную окружения PYTHONPATH иначе, чем вы. Синтаксическая ошибка в файлах настройки системной оболочки может испортить значение PYTHONPATH, даже если она кажется вам нормальной. Например, в Windows при наличии пробелов вокруг = в команде DOS set в файле с настройками (например, set NAME = VALUE) в переменную NAME в действительности будет записана пустая строка, а не VALUE!

нуждены были предусмотреть динамическую настройку пути поиска, так как применение веб-сервера не позволяет делать какие-либо предположения о путях к импортируемым модулям.

Пути к каталогам в Windows

Обратите внимание, что при настройке списка `sys.path` в примерах выше были использованы литералы необрабатываемых строк (raw string): поскольку обратный слеш в строке Python обычно означает начало экранированной последовательности, пользователи Windows должны следить за тем, чтобы удваивать символы слеша при использовании в строках с путями к каталогам (например, в строке `"C:\\dir"` комбинация `\\` в действительности является экранированной последовательностью, означающей символ `\`), или использовать константы необрабатываемых строк, чтобы иметь возможность вставлять символ обратного слеша без всяких ухищрений (например, `r"C:\dir"`).

При просмотре путей к каталогам в Windows (например, при выводе списка `sys.path` в интерактивной оболочке) Python выводит `\\`, как один символ `\`. Формально можно обойтись одним символом `\`, если за ним следует символ, не воспринимаемый Python как продолжение экранированной последовательности, но использовать удвоение и необрабатываемые строки обычно легче, чем запоминать таблицы экранированных последовательностей.

Обратите также внимание, что большинство библиотечных функций Python в качестве разделителей элементов путей к каталогам принимают как прямой (`/`), так и обратный (`\`) слеш, 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.

Таблица загруженных модулей

В модуле `sys` есть также средства, позволяющие подключиться к интерпретатору. Например, переменная `sys.modules` служит словарем, содержащим записи вида *name:module* для каждого модуля, импортированного в сеанс или программу Python (точнее, в вызывающий процесс Python):

```
>>> sys.modules
{'reprlib': <module 'reprlib' from 'c:\python31\lib\reprlib.py'>,
...часть строк удалена...

>>> list(sys.modules.keys())
```

```
['reprlib', 'heapq', '__future__', 'sre_compile', '_collections', 'locale',
'_sre', 'functools', 'encodings', 'site', 'operator', 'io', '__main__', ...часть
строк удалена... ]

>>> sys
<module 'sys' (built-in)>
>>> sys.modules['sys']
<module 'sys' (built-in)>
```

Мы могли бы использовать эту переменную для создания программ, выводящих или иным образом обрабатывающих все модули, загруженные программой (нужно просто обойти в цикле список ключей `sys.modules`).

Аналогичным средством подключения к интерпретатору является счетчик ссылок на объекты, доступный через переменную `sys.getrefcount`, и список имен модулей, встроенных в выполняемый файл интерпретатора Python (`sys.builtin_module_names`). Более подробные сведения вы найдете в руководстве по библиотеке Python. Подобные переменные главным образом предназначены для получения внутренней информации интерпретатора Python, но иногда они могут иметь большое значение для программистов, создающих инструменты для других программистов.

Сведения об исключениях

Некоторые атрибуты модуля `sys` позволяют получить все сведения о самом последнем исключении, возбужденном интерпретатором Python. Это удобно, когда требуется реализовать обобщенную процедуру обработки исключений. Например, функция `sys.exc_info` возвращает кортеж, содержащий тип последнего исключения, его значение и объект с трассировочной информацией. В модели исключений, которая в Python 3 полностью основана на классах, первые два элемента кортежа соответствуют классу последнего исключения и его экземпляру:

```
>>> try:
...     raise IndexError
... except:
...     print(sys.exc_info())
...
(<class 'IndexError'>, IndexError(), <traceback object at 0x019B8288>)
```

Эту информацию можно использовать для создания собственного сообщения об ошибке, выводимого в окне диалога графического интерфейса или в веб-странице HTML (напомню, что не обработанные исключения по умолчанию завершают программы с выводом сообщения об ошибке). Первые два элемента кортежа, возвращаемого этой функцией, по умолчанию предусматривают вывод полезной информации, а третий элемент, объект с трассировочной информацией, можно обработать с помощью стандартного модуля `traceback`:

```
>>> import traceback, sys
>>> def grail(x):
...     raise TypeError('already got one')
...
>>> try:
...     grail('arthur')
... except:
...     exc_info = sys.exc_info()
...     print(exc_info[0])
...     print(exc_info[1])
...     traceback.print_tb(exc_info[2])
...
<class 'TypeError'>
already got one
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in grail
```

Модуль `traceback` может также представлять сообщения в виде строк и записывать их в указанный объект файла – более подробную информацию вы найдете в руководстве по библиотеке Python.

Другие элементы, экспортируемые модулем sys

Модуль `sys` экспортирует и другие часто используемые инструменты, с которыми мы встретимся в контексте более крупных тем и примеров далее в этой части книги. Например:

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

Однако поскольку все эти инструменты ведут к более крупным темам, о них будет рассказано ниже в отдельных разделах.

Модуль os

Как уже говорилось выше, модуль `os` – более крупный из двух основных системных модулей. В нем содержатся все обычные вызовы операционной системы, с которыми вы могли ранее встречаться в своих программах на языке C и в сценариях оболочки. Его вызовы имеют дело с каталогами, процессами, переменными оболочки и так далее. Формально этот модуль предоставляет инструментальные средства POSIX – переносимого стандарта вызовов операционной системы – вместе с платформонезависимыми средствами работы с каталогами, к которым относится вложенный модуль `os.path`. Функционально модуль `os` играет роль переносимого интерфейса к системным вызовам операционной системы: сценарии, написанные с использованием модулей `os` и `os.path`, обычно

могут выполняться на любой платформе без внесения изменений. На некоторых платформах модуль os включает дополнительные инструменты, доступные только на этой платформе (например, низкоуровневые операции с процессами в Unix). Однако, в общем и целом, этот модуль является кросс-платформенным, насколько это технически возможно.

Инструменты в модуле os

Бросим беглый взгляд на основные интерфейсы в модуле os. В табл. 2.1 приводится список наиболее часто используемых инструментов из модуля os, организованных по функциональному назначению.

Таблица 2.1. Часто используемые инструменты из модуля os

Область применения	Инструменты
Переменные окружения	os.environ
Запуск программ	os.system, os.popen, os.execv, os.spawnv
Порождение дочерних процессов	os.fork, os.pipe, os.waitpid, os.kill
Дескрипторы файлов, блокировки	os.open, os.read, os.write
Обработка файлов	os.remove, os.rename, os.mkfifo, os.mkdir, os.rmdir
Инструменты администрирования	os.getcwd, os.chdir, os.chmod, os.getpid, os.listdir, os.access
Инструменты обеспечения переносимости	os.sep, os.pathsep, os.curdir, os.path.split, os.path.join
Инструменты для работы с путями к каталогам	os.path.exists('path'), os.path.isdir('path'), os.path.getsize('path')

Если попробовать получить перечень атрибутов этого модуля в интерактивном режиме, получится громадный список имен, который будет различным для разных версий Python. Скорее всего, он будет зависеть от платформы и не будет слишком полезен, если не изучить, что означает каждое имя (я позволил себе немного отформатировать этот список и удалить часть строк для экономии места – запустите эту команду у себя):

```
>>> import os
>>> dir(os)
['F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_
NOINHERIT', 'O_RANDOM', 'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED',
'O_TEMPORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_
NOWAITO', 'P_OVERLAY', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET',
'TMP_MAX',
...здесь было удалено 9 строк...
'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 'remove', 'rem
ovedirs', 'rename', 'renames', 'rmdir', 'sep', 'spawnl', 'spawnle', 'spawnv',
'spawnve', 'startfile', 'stat', 'stat_float_times', 'stat_result', 'statvfs_
```

```
result', 'strerror', 'sys', 'system', 'times', 'umask', 'unlink', 'urandom',
'utime', 'waitpid', 'walk', 'write']
```

Помимо этого вложенный модуль `os.path` экспортирует дополнительные инструменты, большинство из которых связано с обеспечением переносимости при выполнении операций с именами файлов и каталогов:

```
>>> dir(os.path)
['__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
'_get_altsep', '_get_bothseps', '_get_colon', '_get_dot', '_get_empty', '_
get_sep', '_getfullpathname', 'abspath', 'altsep', 'basename', 'commonprefix',
'curdir', 'defpath', 'devnull', 'dirname', 'exists', 'expanduser', 'expandvars',
'extsep', 'genericpath', 'getatime', 'getctime', 'getmtime', 'getsize',
'isabs', 'isdir', 'isfile', 'islink', 'ismount', 'join', 'lexists', 'normcase',
'normpath', 'os', 'pardir', 'pathsep', 'realpath', 'relpath', 'sep', 'split',
'splitdrive', 'splitext', 'splitunc', 'stat', 'supports_unicode_filenames',
'sys']
```

Средства администрирования

Если вы полагаете, что приведения этих огромных листингов недостаточно, чтобы двинуться дальше, давайте поэкспериментируем с некоторыми из простейших инструментов модуля `os` в интерактивной оболочке. Как и модуль `sys`, модуль `os` содержит набор инструментов для получения информации и администрирования:

```
>>> os.getpid()
7980
>>> os.getcwd()
'C:\\PP4thEd\\Examples\\PP4E\\System'

>>> os.chdir(r'C:\Users')
>>> os.getcwd()
'C:\\Users'
```

Здесь видно, что функция `os.getpid` возвращает числовой идентификатор (ID) процесса (уникальный идентификатор выполняющейся программы, определяемый системой), а функция `os.getcwd` возвращает текущий рабочий каталог. Текущим рабочим каталогом является тот, в котором предполагается нахождение файлов, открываемых сценарием, если в их именах явно не указан путь к каталогу. Вот почему ранее я предлагал запустить следующую команду именно в том каталоге, где находится файл *more.py*:

```
C:\...\PP4E\System> python more.py more.py
```

Аргумент с именем входного файла здесь задан без явного пути к каталогу (хотя его можно указать, чтобы обеспечить постраничный вывод файлов из другого каталога). Если потребуется назначить текущим другой каталог, вызовите функцию `os.chdir`: ваш программный код будет выполняться относительно нового каталога до конца программы (или нового вызова `os.chdir`). В следующей главе еще будет говориться

о понятии текущего рабочего каталога и его связи с операцией импортирования модулей, когда мы будем изучать контекст выполнения сценария.

Константы переносимости

Модуль os экспортирует также ряд имен, упрощающих программирование для нескольких платформ. В этот набор входят специфические для платформ настройки символов-разделителей компонентов путей и каталогов, указателей на родительский и текущий каталоги и символов, используемых для завершения строк в используемой системе:

```
>>> os.pathsep, os.sep, os.pardir, os.curdir, os.linesep
(';', '\\', '..', '.', '\r\n')
```

Константа os.sep определяет символ, который используется в качестве разделителя компонентов пути к каталогу на платформе, где выполняется Python. Она автоматически получает значение \ в Windows, / – в POSIX-совместимых системах и : – в некоторых версиях Mac. Аналогично константа os.pathsep определяет символ, отделяющий каталоги в списках каталогов. Она получает значение : в POSIX-совместимых системах и ; – в DOS и Windows.

Использование таких атрибутов для составления и разбора относящихся к системе строк делает сценарии полностью переносимыми. Например, вызов вида dirpath.split(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 в некоторых версиях на Mac. Как уже говорилось выше, при определении имен открываемых файлов в Windows допускается использовать символы прямого слеша вместо символов обратного слеша, но применение этих констант обеспечивает независимость программного кода, реализующего операции с каталогами, от платформы, на которой он выполняется.

Обратите также внимание, что функция os.linesep в примере выше возвращает последовательность символов \r\n – экранированные последовательности, соответствующие комбинации символов возврата каретки и перевода строки, которая в Windows используется как признак конца строки и на которую обычно никто не обращает внимания при обработке текстовых файлов в Python. Подробнее о преобразовании символов конца строки будет рассказываться в главе 4.

Основные инструменты os.path

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

```
>>> os.path.isdir(r'C:\Users'), os.path.isfile(r'C:\Users')
(True, False)
>>> os.path.isdir(r'C:\config.sys'), os.path.isfile(r'C:\config.sys')
(False, True)
>>> os.path.isdir('nonesuch'), os.path.isfile('nonesuch')
(False, False)

>>> os.path.exists(r'c:\Users\Brian')
False
>>> os.path.exists(r'c:\Users\Default')
True
>>> os.path.getsize(r'C:\autoexec.bat')
24
```

Функции os.path.isdir и os.path.isfile сообщают нам о том, является ли имя файла каталогом или простым файлом; обе они возвращают False, если указанный файл не существует (то есть отсутствие файла предполагает отрицание). Есть также функции для разбиения или объединения строк путей к каталогам, которые автоматически используют соглашения об именовании каталогов для 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 в Windows
>>> os.path.dirname(name), os.path.basename(name)
('C:\\temp', 'data.txt')

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

>>> os.path.splitext(r'C:\PP4thEd\Examples\PP4E\PyDemos.pyw')
('C:\\PP4thEd\\Examples\\PP4E\\PyDemos', '.pyw')
```

Функция os.path.split отделяет имя файла от пути к его каталогу, а os.path.join снова соединяет их вместе, и все это – совершенно переносимым образом, с использованием соглашений по оформлению путей, действующих в той системе, где они вызываются. Функции dirname и basename возвращают первый и второй элементы, возвращаемые функцией split, и реализованы просто для удобства, а функция splitext отделяет расширение файла (за последним символом .). Тонкое замечание: эти функции по своему действию почти эквивалентны строковым методам split и join, если вызывать их относительно строковой константы os.sep. Почти, но не совсем:

```
>>> os.sep
'\\'
>>> pathname = r'C:\PP4thEd\Examples\PP4E\PyDemos.pyw'

>>> os.path.split(pathname)                # отделить имя файла от каталога
('C:\\PP4thEd\\Examples\\PP4E', 'PyDemos.pyw')

>>> pathname.split(os.sep)                 # разбить путь по символам слеша
['C:', 'PP4thEd', 'Examples', 'PP4E', 'PyDemos.pyw']

>>> os.sep.join(pathname.split(os.sep))
'C:\\PP4thEd\\Examples\\PP4E\\PyDemos.pyw'

>>> os.path.join(*pathname.split(os.sep))
'C:PP4thEd\\Examples\\PP4E\\PyDemos.pyw'
```

Последний вызов `join` требует передачи отдельных аргументов (отсюда и символ `*`), но он не вставляет первый символ слеша после буквы, обозначающей имя диска в Windows. Если подобные отличия имеют большое значение, используйте предшествующий вызов метода `str.join`. Функция `normpath` может пригодиться в ситуациях, когда в путях произвольно смешиваются разделители компонентов пути для Unix и Windows:

```
>>> mixed
'C:\\temp\\public/files/index.html'
>>> os.path.normpath(mixed)
'C:\\temp\\public\\files\\index.html'
>>> print(os.path.normpath(r'C:\temp\\sub\.\file.ext'))
C:\temp\sub\file.ext
```

В этом модуле имеется также функция `abspath`, которая переносимым образом возвращает полное имя файла. Она учитывает добавленный текущий каталог, родительский каталог `..` и многое другое:

```
>>> os.chdir(r'C:\Users')
>>> os.getcwd()
'C:\\Users'
>>> os.path.abspath('')      # пустая строка означает тек. раб. каталог (cwd)
'C:\\Users'

>>> os.path.abspath('temp')  # расширяет до полного пути к файлу в тек. кат.
'C:\\Users\\temp'
>>> os.path.abspath(r'PP4E\dev') # частичный путь относительно тек. раб. кат.
'C:\\Users\\PP4E\\dev'

>>> os.path.abspath('.')     # расширяет относительные пути
'C:\\Users'
>>> os.path.abspath('..')
'C:\\'
```

```
>>> os.path.abspath(r'..\examples')
'C:\\examples'

>>> os.path.abspath(r'C:\PP4thEd\chapters') # абсолютные пути не изменяются
'C:\\PP4thEd\\chapters'
>>> os.path.abspath(r'C:\temp\spam.txt')
'C:\\temp\\spam.txt'
```

Поскольку имена файлов считаются относящимися к текущему рабочему каталогу, если не заданы полными путями, функция `os.path.abspath` может пригодиться, если потребуется показать пользователю, какой каталог используется в действительности для сохранения файла. В Windows, например, при запуске программ с графическим интерфейсом щелчком на ярлыках в проводнике или на рабочем столе рабочим каталогом программы является тот, в котором находится запускаемый файл, что не всегда очевидно пользователю. В таких случаях может помочь вывод значения, возвращаемого функцией `abspath` для файла.

Выполнение команд оболочки из сценариев

Модуль `os` позволяет также выполнять команды оболочки из сценариев Python. Это понятие переплетается с другими, такими как потоки ввода-вывода, которые не будут освещаться в полной мере до следующей главы, но, поскольку это одно из ключевых понятий, которое будет использоваться на протяжении всей данной части книги, бегло коснемся основ. В модуле `os` имеются две функции, позволяющие запустить из сценария любую команду, которую можно ввести в окне консоли:

`os.system`

: Запускает команду оболочки из сценария Python

`os.popen`

: Запускает команду оболочки и соединяется с ее потоками ввода или вывода

Кроме того, существует относительно новый модуль `subprocess`, обеспечивающий более точное управление запускаемыми командами оболочки через потоки ввода-вывода, который может использоваться как альтернатива и даже для реализации двух функций, представленных выше (хотя и за счет некоторой сложности программного кода).

Что такое команда оболочки?

Чтобы понять область действия этих вызовов, нужно сначала ввести несколько терминов. В данной книге под термином *оболочка* (*shell*) подразумевается система, которая считывает и выполняет командные строки на вашем компьютере, а под *командой оболочки* (*shell command*) подразумевается командная строка, которую вы обычно вводите в ответ на приглашение оболочки.

В Windows, например, можно открыть окно консоли MS-DOS (она же Командная строка (Command Prompt)) и вводить в нем команды DOS, такие как dir для получения списка каталогов, type для просмотра файла, имена программ, которые нужно запустить, и так далее. DOS является системной оболочкой, а команды, такие как dir и type, – командами оболочки. В Linux и Mac OS X можно запустить новый сеанс оболочки, открыв окно терминала, и также вводить в него команды оболочки – ls для вывода списка каталогов, cat для просмотра файлов и так далее. Для Unix существует множество оболочек (например, csh, ksh), но все они читают и выполняют командные строки. Ниже показаны две команды оболочки, введенные и выполненные в окне консоли MS-DOS под Windows:

```
C:\...\PP4E\System> dir /B      ...ввод команды оболочки
helloshell.py                   ...далее следует вывод этой команды
more.py                         ...DOS играет роль оболочки в Windows
more.pyc
spam.txt
__init__.py

C:\...\PP4E\System> type helloshell.py
# a Python program
print('The Meaning of Life')
```

Выполнение команд оболочки

Конечно, все это не имеет прямого отношения к Python (несмотря на то, что сценарии командной строки на языке Python иногда ошибочно называют инструментами оболочки). Но, поскольку функции system и popen из модуля os позволяют сценариям Python выполнять любые команды, понятные оболочке системы, мы можем использовать в своих сценариях любые имеющиеся в системе инструменты командной строки, независимо от того, написаны они на Python или нет. Например, ниже приводится некоторый программный код на языке Python, который выполняет две команды оболочки DOS, введенные выше в ответ на приглашение оболочки:

```
C:\...\PP4E\System> python
>>> import os
>>> os.system('dir /B')
helloshell.py
more.py
more.pyc
spam.txt
__init__.py
0
>>> os.system('type helloshell.py')
# a Python program
print('The Meaning of Life')
0
```

```
>>> os.system('type hellshell.py')
The system cannot find the file specified.
1
```

Нули, которые выводятся по окончании выполнения первых двух команд, являются значениями, возвращаемыми самой функцией `system`. Функцию `system` можно использовать для выполнения любой командной строки, которую допускается ввести в ответ на приглашение оболочки (здесь приглашением является `C:\...\PP4E\System>`). Выводимые командой данные обычно попадают в стандартный поток вывода сеанса Python или программы.

Обмен данными с командами оболочки

Но что если в сценарии потребуется перехватить данные, выводимые командой? Функция `os.system` просто запускает команду оболочки, тогда как функция `os.popen` дополнительно соединяется со стандартными потоками ввода-вывода команды, – обратно возвращается объект, подобный файлу, по умолчанию соединенный с выводом команды (если передать функции `popen` флаг режима `w`, то вместо этого произойдет подключение к потоку ввода команды). Используя этот объект для чтения данных, выводимых командой, запущенной с помощью `popen`, можно перехватывать текст, в обычных условиях появляющийся в окне консоли, где вводится команда:

```
>>> open('helloshell.py').read()
"# a Python program\nprint('The Meaning of Life')\n"

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

>>> listing = os.popen('dir /B').readlines()
>>> listing
['helloshell.py\n', 'more.py\n', 'more.pyc\n', 'spam.txt\n', '__init__.py\n']
```

Здесь мы получаем содержимое файла сначала обычным способом (средствами Python для работы с файлами), а затем – как вывод команды оболочки `type`. Чтение вывода команды `dir` позволяет получить список файлов в каталоге, который затем можно обработать в цикле. В главе 4 будут представлены другие способы получения такого списка, и там же мы познакомимся с *итераторами* файлов, которые в большинстве программ делают ненужным вызов функции `readlines`, показанный в примере выше, за исключением отображения списка в интерактивной оболочке, как в данном случае (дополнительная информация по этой теме приводится также во врезке «subprocess, os.popen и итераторы» ниже).

До сих пор мы выполняли простые команды DOS, но поскольку эти функции могут запускать любые допустимые команды, с их помощью можно также запускать другие сценарии Python. Предположим, что

каталог с выполняемым файлом интерпретатора Python в вашей системе находится в пути поиска файлов (чтобы можно было использовать короткую команду «python» вместо «C:\Python31\python»):

```
>>> os.system('python helloshell.py')  # запустит программу на языке Python
The Meaning of Life
0
>>> output = os.popen('python helloshell.py').read()
>>> output
'The Meaning of Life\n'
```

Во всех этих примерах командные строки, передаваемые функциям system и popen, жестко «зашиты» непосредственно в программный код, но нет никаких причин, по которым программы на языке Python не могли бы создавать такие строки на этапе выполнения с помощью обычных строковых операций (+, % и других). Учитывая, что команды могут конструироваться и выполняться динамически, функции system и popen превращают сценарии на языке Python в гибкие и переносимые средства запуска и управления другими программами. Например, тестовый «управляющий» сценарий на языке Python можно использовать для запуска программ, написанных на любых языках программирования (например, C++, Java, Python), и анализа их вывода. Такой сценарий будет рассмотрен в главе 6. В следующей главе мы снова вернемся к функции os.popen, где будем рассматривать ее в соединении с проблемой перенаправления потоков ввода-вывода – как будет показано там, механизм перенаправления также может использоваться для передачи ввода в программы.

Альтернатива на основе модуля subprocess

Как уже говорилось, в последних версиях Python появился модуль subprocess, позволяющий добиться того же эффекта, что и функции os.system и os.popen. Вообще говоря, для этого придется написать дополнительный программный код, но этот модуль обеспечивает более полный контроль над подключением и использованием потоков ввода-вывода. Это особенно полезно для реализации сложных схем связывания потоков ввода-вывода.

Например, чтобы запустить простую команду оболочки, как мы делали это с помощью функции os.system выше, можно воспользоваться функцией call из нового модуля, которая действует похожим образом (чтобы запустить такую команду, как type, встроенную в оболочку Windows, требуется соблюсти дополнительные условия, хотя для запуска обычных выполняемых файлов, таких как python, этого не требуется):

```
>>> import subprocess
>>> subprocess.call('python helloshell.py')        # напоминает os.system()
The Meaning of Life
0
>>> subprocess.call('cmd /C "type helloshell.py"') # встроенная команда
```

```
# a Python program
print('The Meaning of Life')
0
>>> subprocess.call('type helloshell.py', shell=True) # альтернативный способ
# a Python program                                    # для встроенных команд
print('The Meaning of Life')
0
```

Обратите внимание на аргумент `shell=True` в последнем вызове. Это платформозависимая особенность:

- Чтобы запустить встроенную команду оболочки в Windows, инструментам модуля `subprocess`, таким как `call` и `Popen` (об этой функции будет рассказываться ниже), необходимо передавать аргумент `shell=True`. Команды Windows, такие как `type`, требуют соблюдения дополнительных условий, но для запуска обычных выполняемых файлов, таких как `python`, этого не требуется.
- В Unix-подобных платформах, когда аргумент `shell` принимает значение `False` (по умолчанию), команда запускается непосредственно вызовом функции `os.execvp`, с которой мы встретимся в главе 5. Если в этом аргументе передать `True`, команда будет выполнена с помощью оболочки, при этом вы можете указать используемую оболочку в дополнительном аргументе.

Подробнее о некоторых из этих особенностей мы поговорим ниже, а пока достаточно будет запомнить, что в Unix-подобных системах вам может потребоваться передавать аргумент `shell=True` в некоторых примерах в этом разделе и в книге, если они предполагают использование таких особенностей оболочки, как путь поиска программ. Поскольку я запускаю примеры в Windows, этот аргумент я часто буду опускать.

Помимо имитации функции `os.system`, мы точно так же можем использовать этот модуль для имитации функции `os.popen`, чтобы запускать команды оболочки и получать ее вывод в нашем сценарии:

```
>>> pipe = subprocess.Popen('python helloshell.py', stdout=subprocess.PIPE)
>>> pipe.communicate()
(b'The Meaning of Life\r\n', None)
>>> pipe.returncode
0
```

Здесь мы связали поток стандартного вывода команды оболочки с каналом и вызвали метод `communicate`, ожидающий завершения команды и принимающий текст, который она выводит в стандартный поток вывода и в стандартный поток ошибок. Код завершения команды доступен в виде атрибута, после того как она будет выполнена. Точно так же мы могли бы использовать отдельную функцию чтения потока стандартного вывода команды и отдельную функцию ожидания ее завершения (которая возвращает код завершения):

```
>>> pipe = subprocess.Popen('python helloshell.py', stdout=subprocess.PIPE)
>>> pipe.stdout.read()
b'The Meaning of Life\r\n'
>>> pipe.wait()
0
```

Фактически существует возможность прямой замены вызова `os.popen` объектом `subprocess.Popen`:

```
>>> from subprocess import Popen, PIPE
>>> Popen('python helloshell.py', stdout=PIPE).communicate()[0]
b'The Meaning of Life\r\n'
>>>
>>> import os
>>> os.popen('python helloshell.py').read()
'The Meaning of Life\n'
```

Как видите, реализация относительно простых случаев с помощью модуля `subprocess` требует дополнительной работы. Но ситуация меняется в лучшую сторону, когда возникает необходимость гибкого управления потоками ввода-вывода. Фактически, благодаря возможности обрабатывать стандартные потоки вывода и ошибок команды похожими способами, модуль `subprocess` в Python 3.X заменил оригинальные функции `os.popen2`, `os.popen3` и `os.popen4`, имевшиеся в Python 2.X. Теперь эти функции являются лишь частными случаями использования интерфейса объектов модуля `subprocess`. Поскольку в более сложных случаях использования этого модуля предполагается взаимодействие со стандартными потоками ввода-вывода, мы отложим дальнейшее обсуждение этого модуля, пока не познакомимся с механизмом перенаправления потоков в следующей главе.

Ограничения, присущие командам оболочки

Прежде чем двинуться дальше, вы должны запомнить два ограничения, связанные с функциями `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. Предыдущие примеры с командами DOS `dir` и `type`, например, работают только в Windows, а для Unix-подобных платформ их следует изменить так, чтобы они выполняли команды `ls` и `cat`.

Во-вторых, важно помнить, что запуск файлов Python как самостоятельных программ таким способом очень отличается от импорта программных файлов и вызова функций, объявленных в них, и обычно происходит гораздо медленнее. Когда вызываются функции `os.system` и `os.popen`, им приходится запускать совершенно новую и независимую программу, выполняемую операционной системой (как правило, они запускают команды в виде новых процессов). При импорте программы в качестве модуля интерпретатор Python просто загружает и выполня-

ет код файла в том же процессе, генерируя объект модуля. При этом никакие другие программы не запускаются.[1]

Могут быть веские основания для построения системы в виде отдельных программ, и в следующей главе будут рассмотрены такие темы, как аргументы командной строки и потоки ввода-вывода, которые позволяют программам передавать информацию в обоих направлениях. Но в большинстве случаев операция импортирования является более быстрым и прямым способом построения систем.

Если вы серьезно намерены использовать эти вызовы, то следует знать, что вызов `os.system` обычно блокирует (то есть приостанавливает) вызывающую программу до завершения запущенной ею команды. В Linux и Unix-подобных платформах имеется возможность заставить команду выполняться независимо и параллельно с вызвавшей ее программой, добавив в конец командной строки оператор `&` выполнения в фоновом режиме:

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

В Windows запуск с помощью команды DOS `start` обычно также приводит к запуску команды в виде независимого процесса, выполняющегося параллельно:

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

В действительности, такой способ запуска команд оказался настолько удобным, что в последние версии Python была добавлена функция `os.startfile`. Эта функция открывает файл с помощью программы, указанной в реестре Windows для файлов этого типа, как если бы был выполнен щелчок мышью на ярлыке этого файла:

```
os.startfile("webpage.html") # open file in your web browser
os.startfile("document.doc") # open file in Microsoft Word
os.startfile("myscript.py")  # run file with Python
```

---

[1] Инструкция на языке Python `exec(open(file).read())` тоже выполняет код программного файла, но внутри того же процесса, который ее вызвал. В этом отношении она похожа на операцию импортирования, но по своему действию она больше похожа на операцию *вставки* содержимого файла в вызывающую программу в том месте, где стоит вызов `exec` (если явно не передаются словари глобального или локального пространства имен). В отличие от операции импортирования, функция `exec` читает и выполняет программный код файла без всяких проверок (один и тот же файл можно выполнить несколько раз в одном процессе); при выполнении файла не создается объект модуля; и, если функции явно не передаются словари пространств имен, операции присваивания в программном коде файла могут затирать переменные в области видимости, где находится вызов `exec`. За дополнительными подробностями обращайтесь к руководству по библиотеке Python.

Функция os.popen обычно не блокирует вызывающую программу (она, по определению, должна иметь возможность читать или писать в возвращаемый объект файла). Тем не менее вызывающая программа иногда все же может оказаться заблокированной в любой операционной системе, Windows или Linux, если объект канала будет закрыт до завершения порожденной программы (например, при сборке мусора) или когда канал будет прочитан до исчерпания (например, с помощью метода read() канала). Как будет показано далее в этой части книги, для параллельного исполнения программ без блокирования можно использовать функции os.fork/exec в Unix и os.spawnv – в Windows.

Поскольку функции system и popen из модуля os, а также модуль subprocess попадают еще и в категорию средств запуска программ, перенаправления потоков ввода-вывода и средств взаимодействия процессов, они снова появятся в последующих главах, поэтому пока мы отложим их дальнейшее обсуждение. Если вам уже сейчас необходимы дополнительные подробности, прочитайте раздел, касающийся вопросов перенаправления потоков ввода-вывода в следующей главе, и раздел, описывающий получение списка каталогов, в главе 4.

Другие элементы, экспортируемые модулем os

Мы рассмотрели в модуле os, что смогли. 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 приводится краткий справочный обзор. Среди прочего на вооружении модуля os состоят:

os.environ

Извлекает и устанавливает значения переменных окружения оболочки.

os.fork

Запускает новый дочерний процесс в Unix-подобных системах.

os.pipe

Обеспечивает обмен данными между программами.

os.execlp

Запускает новые программы.

os.spawnv

Запускает новые программы с возможностью низкоуровневого управления.

os.open

Открывает файл с использованием файлового дескриптора.

`os.mkdir`

Создает новый каталог.

`os.mkfifo`

Создает новый именованный канал.

`os.stat`

Получает низкоуровневую информацию о файле.

`os.remove`

Удаляет файл по строке пути к нему.

`os.walk`

Применяет функцию или тело цикла ко всем элементам в дереве каталогов.

И так далее. Заранее предупреждаю: модуль `os` предоставляет группу функций для открытия, чтения и записи файлов, но все они используют доступ к файлам на низком уровне и кардинально 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 (например, когда требуется открыть файл, заблокировав доступ к нему из других программ).

В следующей главе мы будем использовать инструменты из модулей `sys` и `os` для решения обычных системных задач, но объем данной книги не позволяет приводить полные списки содержимого встречающихся модулей. Если вы этого еще не сделали, ознакомьтесь с содержимым таких модулей, как `os` и `sys`, обратившись к ресурсам, описанным выше. А теперь перейдем к исследованию дополнительных системных инструментов в контексте более широких понятий системного программирования.

subprocess, os.popen и итераторы

В главе 4 мы будем исследовать итераторы файлов, но перед тем, как взять эту книгу, вы наверняка уже ознакомились с основами. Поскольку объекты, возвращаемые функцией `os.popen`, обладают итераторами, позволяющими читать данные по одной строке за раз, использование метода `readlines` этих объектов обычно является излишним. Например, ниже приводится пример чтения строк, которые выводятся другой программой, без явного использования методов чтения:

```
>>> import os
>>> for line in os.popen('dir /B *.py'): print(line, end='')
...
```

```
helloshell.py
more.py
__init__.py
```

Интересно, что в Python 3.1 функция os.popen реализована с использованием объекта subprocess.Popen, с которым мы познакомились в этой главе. Вы можете убедиться в этом, заглянув в файл os.py в стандартной библиотеке Python (в Windows вы найдете этот файл в каталоге *C:\Python31\Lib*). Результатом вызова функции os.popen является объект, управляющий объектом Popen и его каналами:

```
>>> I = os.popen('dir /B *.py')
>>> I
<os._wrap_close object at 0x013BC750>
```

Этот объект-обертка канала определяет метод __iter__, поэтому он поддерживает возможность итераций по строкам, которые могут выполняться автоматически (например, в цикле for, как показано выше) или вручную. Любопытно отметить, что, несмотря на наличие в объекте-обертке поддержки прямого вызова метода __next__, как если бы этот объект обладал собственным итератором (подобно простым файлам), тем не менее он не поддерживает встроенную функцию next, хотя последняя, вероятно, просто вызывает метод __next__:

```
>>> I = os.popen('dir /B *.py')
>>> I.__next__()
'helloshell.py\n'

>>> I = os.popen('dir /B *.py')
>>> next(I)
TypeError: _wrap_close object is not an iterator
```

Причина такого поведения скрыта глубоко в недрах реализации – прямой вызов метода __next__ перехватывается методом __getattr__, определенном в объекте-обертке канала, и преобразуется в вызов метода обернутого объекта. Но функция next обращается к механизму перегрузки операторов, который в Python 3.X действует в обход метода __getattr__, когда производится вызов методов со специальными именами, такими как __next__. Поскольку объект-обертка канала не определяет собственный метод __next__, обращение к нему не перехватывается и не передается обернутому объекту, что приводит к ошибке при вызове встроенной функции next. Как детально объясняется в книге «Изучаем Python», метод __getattr__ обертки не вызывается по той простой причине, что в Python 3.X поиск методов начинается не с экземпляра, а с класса.

Такое поведение может быть или не быть ожидаемым, но вам не придется беспокоиться об этом при выполнении итераций по строкам в канале с помощью цикла for, генераторов и других инструментов. Тем не менее, чтобы обеспечить выполнение итераций вручную, необходимо сначала вызвать встроенную функцию iter — она вызовет метод __iter__ объекта-обертки канала и обеспечит корректную поддержку обоих способов перемещения по строкам:

```
>>> I = os.popen('dir /B *.py')
>>> I = iter(I)          # так поступает цикл for
>>> I.__next__()         # теперь обе формы итераций действуют правильно
'helloshell.py\n'
>>> next(I)
'more.py\n'
```

3

Контекст выполнения сценариев

«Ваши аргументы, пожалуйста!»

Сценарии на языке Python выполняются не в вакууме (хотя, возможно, вы слышали что-то иное). В зависимости от платформы и процедуры запуска программы на языке Python выполняются в определенном контексте, то есть обладают информацией, которая автоматически передается операционной системой при запуске программы. Например, у сценариев есть доступ к следующим видам входных данных системного уровня и интерфейсов:

Текущий рабочий каталог

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

Аргументы командной строки

`sys.argv` предоставляет доступ к словам в команде, с помощью которой была запущена программа и которые играют роль входных данных сценария.

Переменные оболочки

`os.environ` предоставляет интерфейс к переменным окружения, созданным в охватывающей оболочке (или родительской программой) и передаваемым сценарию.

Стандартные потоки ввода-вывода

`sys.stdin`, `stdout` и `stderr` экспортируют три потока ввода-вывода, лежащие в центре инструментов командной оболочки, и могут использоваться в сценариях с помощью функции `print`, `os.popen` и модулем

subprocess, представленными в главе 2, с помощью класса io.StringIO и других инструментов.

Такие инструменты могут играть роль входных данных сценариев, параметров настройки и так далее. В этой главе мы исследуем все эти четыре инструмента работы с контекстом – их интерфейсы в Python и типичные области их использования.

Текущий рабочий каталог

Понятие текущего рабочего каталога (current working directory, CWD) оказывается ключевым при выполнении некоторых сценариев: это всегда неявно определенное место в файловой системе, где предполагается размещение обрабатываемых сценарием файлов, если в их именах отсутствует абсолютный путь к каталогу. Как мы уже видели, функция os.getcwd позволяет сценарию получить имя текущего рабочего каталога в явном виде, а функция os.chdir позволяет сценарию переместиться в новый текущий рабочий каталог.

Однако имейте в виду, что для имен файлов, не содержащих полного пути, подразумевается, что они находятся в текущем рабочем каталоге, и это не имеет никакого отношения к переменной окружения PYTHONPATH. С технической точки зрения сценарий всегда запускается из текущего рабочего каталога, а не из каталога, содержащего файл сценария. Напротив, при импортировании модулей поиск всегда начинается в каталоге, содержащем сценарий, а не в текущем рабочем каталоге (если только сценарий не размещен в текущем рабочем каталоге). Поскольку это тонкое различие, на котором часто попадаются новички, изучим его более подробно.

Текущий рабочий каталог, файлы и путь поиска модулей

Если запустить сценарий Python, введя в команду, например, python dir1\dir2\file.py, – текущим рабочим каталогом будет каталог, в котором вы находились при вводе этой команды, но не *dirl\dir2*. С другой стороны, Python автоматически добавляет путь к каталогу, где находится сценарий, в начало пути поиска модулей, поэтому *file.py* всегда сможет импортировать другие файлы из *dirl\dir2*, откуда бы он ни был запущен. Чтобы проиллюстрировать это, напишем простой сценарий, выводящий имя текущего рабочего каталога и путь поиска модулей:

```
C:\...\PP4E\System> type whereami.py
import os, sys
print('my os.getcwd =>', os.getcwd())  # вывод текущего рабочего каталога
print('my sys.path =>', sys.path[:6])  # вывод первых 6 каталогов в пути поиска
input()                                # ожидает нажатия клавиши
```

Теперь при запуске этого сценария в том каталоге, где он находится, будет выбран ожидаемый текущий рабочий каталог, и имя этого каталога будет добавлено в начало пути поиска. Мы уже встречались со списком `sys.path`, содержащим путь поиска модулей, – первый его элемент может быть пустой строкой, обозначающей текущий рабочий каталог при работе в интерактивной оболочке; здесь большая часть пути к текущему рабочему каталогу усекается до «...» при отображении:

```
C:\...\PP4E\System> set PYTHONPATH=C:\PP4thEd\Examples
C:\...\PP4E\System> python whereami.py
my os.getcwd => C:\...\PP4E\System
my sys.path => ['C:\\...\\PP4E\\System', 'C:\\PP4thEd\\Examples', ...другие
элементы...]
```

Если запускать этот сценарий из других каталогов, вслед за нашим перемещением переместится и текущий рабочий каталог (это каталог, в котором вводятся команды), а Python будет добавлять в начало пути поиска модулей каталог, где находится сам сценарий, что позволит сценарию по-прежнему видеть файлы в своем исходном каталоге. Например, если запустить сценарий, поднявшись на один уровень (`..`), каталог *System* будет добавлен в начало списка `sys.path` и станет первым каталогом, в котором Python станет искать модули, импортируемые сценарием *whereami.py*: первый элемент списка будет нацеливать импорт обратно на каталог, содержащий выполняемый сценарий. Однако поиск файлов, имена которых не содержат полного пути, будет выполняться относительно текущего рабочего каталога (*C:\PP4thEd\Examples\PP4E*), а не в его подкаталоге *System*:

```
C:\...\PP4E\System> cd ..
C:\...\PP4E> python System\whereami.py
my os.getcwd => C:\...\PP4E
my sys.path => ['C:\\...\\PP4E\\System', 'C:\\PP4thEd\\Examples', ...другие
элементы...]

C:\...\PP4E> cd System\temp
C:\...\PP4E\System\temp> python ..\whereami.py
my os.getcwd => C:\...\PP4E\System\temp
my sys.path => ['C:\\...\\PP4E\\System', 'C:\\PP4thEd\\Examples', ...]
```

В результате поиск файлов, имена которых в сценарии не содержат полных путей, будет выполняться в том месте, где была введена *команда* (`os.getcwd`), но операции импортирования по-прежнему будут иметь доступ к каталогу, где находится выполняемый *сценарий* (через первый элемент в списке `sys.path`). Наконец, если файл запускается щелчком на ярлыке, текущим рабочим каталогом станет каталог, содержащий файл, на котором выполнен щелчок. Например, следующие строки будут выведены в новом окне консоли DOS при двойном щелчке на *whereami.py* в проводнике Windows:

```
my os.getcwd => C:\...\PP4E\System
my sys.path => ['C:\\...\\PP4E\\System', ...more... ]
```

В данном случае текущий рабочий каталог, используемый для поиска файлов, и первый каталог в пути поиска модулей совпадают с каталогом, содержащим файл сценария. Обычно все действует так, как предполагается, но здесь вас могут подстерегать две опасности:

- Имена файлов должны содержать полные пути к каталогам, если заранее не известно, из какого каталога будет запущен сценарий.
- Сценарии командной строки не всегда могут использовать текущий рабочий каталог, чтобы увидеть импортируемые файлы, не находящиеся в собственных каталогах сценариев, – для доступа к модулям из других каталогов следует использовать переменную окружения `PYTHONPATH` и пути к импортируемым пакетам.

Например, сценарии из этой книги всегда могут импортировать другие файлы из собственного исходного каталога, не указывая путь к импортируемому пакету (`import file here`), независимо от того, как они запущены. Но, чтобы отыскать файлы в другом месте в дереве каталогов примеров, путь поиска должен проходить через корень пакета *PP4E* (`from PP4E.dir1.dir2 import filethere`), даже если запускать сценарии из каталога, содержащего нужный внешний модуль. Как обычно, чтобы обеспечить возможность импортирования модулей, имя каталога *PP4E\dirl\dir2* можно также добавить в `PYTHONPATH`, чтобы сделать `filethere` видимым отовсюду, без указания пути к импортируемому пакету (хотя лишние каталоги в `PYTHONPATH` увеличивают вероятность конфликта имен). В любом случае импорт всегда осуществляется из исходного каталога сценария или из другого каталога, находящегося в пути поиска Python, а не из текущего рабочего каталога.

Текущий рабочий каталог и командные строки

Это различие между текущим рабочим каталогом и путями поиска модулей 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 C:\...\PP4E\Tools\cleanpyc.py     обработка cwd
```

В данном примере сам файл сценария на языке Python находится в каталоге *C:\...\PP4E\Tools*, но поскольку он запускается из *C:\temp*, то обрабатывает файлы, содержащиеся в *C:\temp* (то есть в текущем рабочем каталоге, а не исходном каталоге сценария). Чтобы обработать с помощью такого сценария файлы, находящиеся где-то в другом месте, нужно просто изменить текущий рабочий каталог с помощью команды `cd` и перейти в каталог, который должен быть обработан:

```
C:\temp> cd C:\PP4thEd\Examples
C:\PP4thEd\Examples> python C:\...\PP4E\Tools\cleanpyc.py    обработка cwd
```

Поскольку текущий рабочий каталог всегда неявно определен, сценарий узнает, какой каталог должен быть обработан, от команды `cd` с той же степенью определенности, что и при явной передаче имени каталога сценарию, как показано ниже (примечание, касающееся переносимости: в других примерах команд вам может потребоваться заключить `*.py` в кавычки, чтобы предотвратить подстановку имен файлов, которая выполняется в некоторых оболочках Unix):

```
C:\...\PP4E\Tools> python find.py *.py C:\temp    обработка указанного каталога
```

В этой командной строке текущим рабочим каталогом является каталог, содержащий запускаемый сценарий, который должен быть выполнен (обратите внимание на отсутствие пути в имени файла сценария). Но поскольку этот сценарий обрабатывает каталог, явно указанный в командной строке (*C:\temp*), в данном случае не имеет никакого значения, какой каталог является текущим рабочим каталогом. Наконец, если потребуется 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 C:\...\PP4E\Tools\find.py *.cxx C:\PP4thEd\Examples\PP4E
```

В этом случае для импорта будут доступны файлы в исходном каталоге сценария *PP4E\Tools*, а обрабатываться будут файлы в каталоге, указанном в командной строке, при этом текущим рабочим каталогом будет совершенно другой каталог (*C:\temp*). Использование последней формы требует больше вводить с клавиатуры, тем не менее в этой книге вам еще не раз встретятся различные текущие рабочие каталоги и командные строки с явными путями к сценариям.

Аргументы командной строки

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

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

Чтобы действительно увидеть аргументы, нужно запустить сценарий из командной строки. В примере 3.1 приводится простой до безобразия сценарий, который всего лишь выводит список `argv` для изучения.

Пример 3.1. PP4E\System\testargv.py

```
import sys
print(sys.argv)
```

Если запустить этот сценарий, он выведет список аргументов командной строки. Обратите внимание, что первым элементом всегда является имя самого выполняемого сценария, независимо от способа запуска (смотрите врезку «Выполняемые сценарии в Unix» далее в этой главе):

```
C:\...\PP4E\System> python testargv.py
['testargv.py']

C:\...\PP4E\System> python testargv.py spam eggs cheese
['testargv.py', 'spam', 'eggs', 'cheese']

C:\...\PP4E\System> python testargv.py -i data.txt -o results.txt
['testargv.py', '-i', 'data.txt', '-o', 'results.txt']
```

Последняя команда в этом фрагменте 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 ключа `-i` является `data.txt` (например, имя входного файла). В качестве параметров командной строки допускается передавать любые слова, но обычно программы накладывают на них некоторые структурные ограничения.

Аргументы командной строки играют в программах такую же роль, как аргументы в функциях: они просто позволяют передать в программу информацию, которая может быть различной для каждого запуска программы. То обстоятельство, что они не определяются жестко в программном коде, позволяет использовать сценарии более универсальными способами. Например, аргументы командной строки могут использоваться для передачи имен файлов сценариям, их обрабатывающим, – взгляните на сценарий *more.py* в главе 2 (пример 2.1), который был нашим первым примером. Другие сценарии могут принимать флаги режима обработки, адреса Интернета и так далее.

Анализ аргументов командной строки

Однако при регулярном использовании аргументов командной строки вы можете обнаружить, что писать код, который вылавливает в списке слова, неудобно. Обычно при запуске программы преобразуют список аргументов в структуры, более удобные для обработки. Ниже приводится один из способов реализации такого преобразования: сценарий в примере 3.2 просматривает список `argv` в поисках пар `-optionname op-`

tionvalue и помещает их в словарь, используя в качестве ключей имена параметров, чтобы потом их было легче извлекать.

Пример 3.2. PP4E\System\testargv2.py

```
"собирает параметры командной строки в словаре"

def getopts(argv):
    opts = {}
    while argv:
        if argv[0][0] == '-':        # поиск пар "-name value"
            opts[argv[0]] = argv[1]  # ключами словарей будут имена параметров
            argv = argv[2:]
        else:
            argv = argv[1:]
    return opts

if __name__ == '__main__':
    from sys import argv             # пример клиентского программного кода
    myargs = getopts(argv)
    if '-i' in myargs:
        print(myargs['-i'])
    print(myargs)
```

Такую функцию можно импортировать и использовать во всех инструментах командной строки. Если запустить этот пример как самостоятельный сценарий, он просто выведет отформатированный словарь аргументов:

```
C:\...\PP4E\System> python testargv2.py
{}

C:\...\PP4E\System> python testargv2.py -i data.txt -o results.txt
data.txt
{'-o': 'results.txt', '-i': 'data.txt'}
```

Естественно, мы могли бы предусмотреть еще более сложную реализацию механизма работы с аргументами, добавить проверку ошибок и тому подобное. Для обработки более сложных командных строк мы могли бы использовать более развитые инструменты обработки командной строки из стандартной библиотеки Python:

- Модуль getopt моделирует поведение одноименной утилиты Unix/C
- Модуль optparse является более современной альтернативой и по общему признанию – более мощной

Оба модуля описаны в руководстве по библиотеке языка Python, и там же вы найдете примеры использования, которые мы не будем приводить здесь ради экономии места. Вообще говоря, чем более широкие возможности по настройке предусматривают ваши сценарии, тем больше труда вам придется вложить в реализацию логики обработки аргументов командной строки.

Выполняемые сценарии в Unix

Пользователям Unix и Linux: текстовые файлы с исходным программным кодом на языке Python можно сделать непосредственно исполняемыми, добавив в их начало особую строку, содержащую путь к интерпретатору Python, и присвоив файлу права на выполнение. Например, сохраните следующий фрагмент в текстовом файле с именем *myscript*:

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

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

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

При запуске таким способом список `sys.argv` по-прежнему будет содержать имя сценария в первом элементе: `["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 исполняемые файлы на самом деле является трюком ОС Unix, а не особенностью Python, но стоит отметить, что можно сделать его несколько менее машинно-зависимым, указав в начале команду `Unix env` вместо пути к исполняемому файлу Python:

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

В этом случае операционная система будет отыскивать интерпретатор Python с помощью значений переменных окружения (обычно с помощью переменной `PATH`). Если один и тот же сценарий потребуется использовать на разных машинах, достаточно будет только изменить на них настройки окружения (а не редактировать исходный программный код сценария). Конечно, файлы Python по-прежнему можно запускать с помощью более явной командной строки:

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

При этом предполагается, что интерпретатор `python` находится в системном пути поиска (иначе нужно указывать полный путь к нему), но этот прием действует на любой платформе, где установлен Python и имеется доступ к командной строке. Поскольку это более переносимый способ, я обычно использую его в примерах книги (за дополнительной информацией по этой теме я рекомендую обращаться к страницам руководства Unix). Но, несмотря на это, особые строки #! 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.

Обратите внимание, что в последних версиях Windows также можно вводить имя сценария непосредственно (без слова *python*), чтобы запустить его, и добавлять строку `#!` в начало сценария не нужно. При установке Python регистрируется в реестре Windows как программа для открытия файлов с расширениями, которые воспринимаются интерпретатором `Python` (*.py* и другие). Это также объясняет, почему сценарии могут запускаться в Windows простым щелчком мыши.

Переменные окружения оболочки

Переменные оболочки, которые иногда называют также переменными окружения, доступны в сценариях на языке Python через `os.environ` – объект Python, напоминающий словарь, в котором для каждой переменной оболочки отводится отдельная запись. Переменные оболочки находятся за пределами интерпретатора Python. Они часто устанавливаются в командной строке, в сценариях начального запуска системы или с помощью панели управления и обычно служат в качестве общесистемных параметров настройки для программ.

На самом деле вам уже должен быть знаком главный пример: путь поиска модулей `PYTHONPATH` является переменной оболочки, которую Python использует при выполнении операции импортирования модулей. После установки этой переменной в системе ее значение становится доступным для каждой запускаемой программы Python. Переменные оболочки могут также устанавливаться программами, чтобы передавать входные данные другим программам в приложении; поскольку обычно их значения наследуются порожденными программами, они могут служить простым средством связи между процессами.

Получение значений переменных оболочки

В Python окружение оболочки является простым предустановленным объектом, для обращения к которому не требуется использовать специальный синтаксис. Операция индексирования объекта `os.environ` строками с именами переменных оболочки (например, `os.environ['USER']`) является эквивалентом знака доллара перед именем переменной в большинстве оболочек Unix (например, `$USER`), использования с двух сторон знака процента в DOS (`%USER%`) и вызова `getenv("USER")` в программе на языке C. Запустим интерактивный сеанс и поэкспериментируем (следующий сеанс выполнялся в Python 3.1 в Windows 7):

```
>>> import os
>>> os.environ.keys()
KeysView(<os._Environ object at 0x013B8C70>)

>>> list(os.environ.keys())
['TMP', 'COMPUTERNAME', 'USERDOMAIN', 'PSMODULEPATH', 'COMMONPROGRAMFILES',
...множество строк было удалено...
'NUMBER_OF_PROCESSORS', 'PROCESSOR_LEVEL', 'USERPROFILE', 'OS', 'PUBLIC',
'QTJAVA']

>>> os.environ['TEMP']
'C:\\Users\\mark\\AppData\\Local\\Temp'
```

Здесь метод `keys` возвращает итерируемый объект со списком установленных переменных, а операция индексирования возвращает значение переменной `TEMP` в Windows. В Linux эти инструкции действуют точно так же, но обычно при запуске Python устанавливаются другие переменные. Поскольку нам знакома переменная `PYTHONPATH`, посмотрим в Python на ее значение и убедимся в его правильности (когда я писал эти строки, в эту переменную временно был добавлен путь к корневому каталогу с примерами к четвертому изданию книги):

```
>>> os.environ['PYTHONPATH']
'C:\\PP4thEd\\Examples;C:\\Users\\Mark\\temp'

>>> for srcdir in os.environ['PYTHONPATH'].split(os.pathsep):
...     print(srcdir)
...
C:\PP4thEd\Examples
C:\Users\Mark\temp

>>> import sys
>>> sys.path[:3]
['', 'C:\\PP4thEd\\Examples', 'C:\\Users\\Mark\\temp']
```

Переменная `PYTHONPATH` содержит строку, содержащую список каталогов, разделенных символом, используемым для разделения таких элементов пути на вашей платформе (например, `;` в DOS/Windows, `:` в Unix и Linux). Чтобы разделить эту строку на составляющие, передадим

строковому методу split разделитель os.pathsep (переносимая константа, дающая правильный разделитель для соответствующей системы). Как обычно, фактический путь поиска, используемый во время выполнения, хранится в списке sys.path и является объединением пути к текущему рабочему каталогу и содержимого переменной окружения PYTHONPATH.

Изменение переменных оболочки

Как и обычные словари, объект os.environ поддерживает обращение по ключу и *присваивание*. Операция присваивания, применяемая к словарям, изменяет значение ключа:

```
>>> os.environ['TEMP']
'C:\\Users\\mark\\AppData\\Local\\Temp
>>> os.environ['TEMP'] = r'c:\temp'
>>> os.environ['TEMP']
'c:\\temp'
```

Но в данном случае выполняются некоторые дополнительные действия. Во всех последних версиях Python значения, присваиваемые ключам os.environ таким способом, автоматически *экспортируются* в другие части приложения. То есть присваивание по ключу изменяет не только объект os.environ в программе, но и соответствующую переменную в окружении *оболочки* для процесса выполняемой программы. Новое значение переменной становится видимым программе на языке Python, всем связанным с ней модулям на языке C и всем программам, порождаемым процессом Python.

За кулисами при присваивании объекту os.environ по ключу происходит вызов os.putenv – функции, изменяющей переменную окружения за границами интерпретатора Python. Чтобы показать, как это работает, нам потребуется пара сценариев, которые изменяют и получают значения переменных оболочки. Первый из них приводится в примере 3.3.

Пример 3.3. PP4E\System\Environment\setenv.py

```
import os
print('setenv...', end=' ')
print(os.environ['USER'])      # выведет текущее значение переменной оболочки

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

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

os.environ['USER'] = 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 приводится в примере 2.5.

Пример 3.4. PP4E\System\Environment\echoenv.py

```
import os
print('echoenv...', end=' ')
print('Hello,', os.environ['USER'])
```

Независимо от способа запуска сценарий *echoenv.py* выводит значение переменной окружения USER. При запуске из командной строки этот сценарий выведет значение, установленное нами в самой оболочке:

```
C:\...\PP4E\System\Environment> set USER=Bob

C:\...\PP4E\System\Environment> python echoenv.py
echoenv... Hello, Bob
```

Однако при запуске из другого сценария, например, из *setenv.py*, с помощью функции os.system или os.popen, с которыми мы познакомились ранее, сценарий *echoenv.py* получит то значение переменной USER, которое было установлено родительской программой:

```
C:\...\PP4E\System\Environment> python setenv.py
setenv... Bob
echoenv... Hello, Brian
echoenv... Hello, Arthur
?Gumby
echoenv... Hello, Gumby

C:\...\PP4E\System\Environment> echo %USER%
Bob
```

Точно так же этот механизм действует и в Linux. Вообще говоря, порождаемая программа всегда *наследует* значения переменных окружения от своих родителей. *Порожденными программами* являются такие, которые запускаются средствами Python, например os.spawnv, комбинацией os.fork/exec в Unix-подобных системах, и os.popen, os.system или с помощью модуля subprocess на ряде других платформ. Все программы, запущенные таким способом, получают значения переменных окружения, существующие в момент запуска в родительском процессе.[1]

[1] Так происходит по умолчанию. Некоторые инструменты запуска программ позволяют сценариям передавать в дочерние программы значения переменных окружения, отличные от своих собственных. Например, функция os.spawnve по своему поведению напоминает функцию os.spawnv, но принимает в виде аргумента дополнительный словарь, представляющий окружение оболочки, которое должно быть передано запускаемой программе. Некоторые разновидности os.exec* (имена которых оканчиваются на «e») тоже принимают явное определение окружения. Подробнее о формате вызова os.exec* рассказывается в главе 5.

Подобный способ установки переменных окружения перед запуском новой программы является одним из способов передачи информации в новую программу. Например, можно написать сценарий, изменяющий переменную PYTHONPATH включением в нее пользовательских каталогов, перед запуском других сценариев. Благодаря этому запущенный сценарий получит свой путь поиска модулей в списке sys.path, потому что переменные оболочки передаются потомкам (такой запускающий сценарий будет представлен в конце главы 6).

Особенности переменных оболочки: родители, putenv и getenv

Обратите внимание на последнюю команду в предыдущем примере – после завершения программы верхнего уровня переменная USER получает свое первоначальное значение. Присвоения значений ключам os.environ передаются за пределы интерпретатора *вниз* по цепочке порожденных программ и никогда не передаются *вверх* процессам родительских программ (включая системную оболочку). Это относится и к программам на языке C, использующим библиотечный вызов putenv, то есть данная особенность не является ограничением, характерным именно для Python.

Это едва ли вызовет проблемы в сценарии Python, являющемся вершиной приложения. Но помните, что настройки оболочки, сделанные внутри программы, действуют, лишь пока выполняется эта программа и порожденные ею дочерние программы. Если вам потребуется экспортировать настройки окружения, чтобы они действовали после завершения программы на языке Python, вам необходимо будет найти платформозависимые расширения, реализующие такую возможность. Попробуйте поискать их на сайте *http://www.python.org* и в Интернете.

Другая тонкость: в нынешней реализации изменение значений в os.environ автоматически приводит к вызову функции os.putenv, которая вызывает функцию putenv в библиотеке языка C, если она доступна на вашей платформе, чтобы экспортировать измененное значение за пределы интерпретатора Python во все связанные с ним расширения на языке C. Однако, хотя изменения в os.environ приводят к вызову os.putenv, тем не менее прямой вызов функции os.putenv не оказывает влияния на содержимое os.environ. По этой причине для изменения окружения предпочтительнее использовать интерфейс os.environ.

Обратите также внимание, что настройки окружения загружаются в os.environ на этапе запуска программы, а не при каждом обращении к этому объекту. По этой причине изменения, выполненные в расширениях на языке C уже после запуска программы, могут не отражаться в os.environ. В языке Python на самом деле имеется более конкретная

функция os.getenv, но она не вызывает функцию getenv из библиотеки языка C, а просто выбирает значения ключей из os.environ в большинстве платформ (во всех в версии 3.X). Для большинства приложений в этом нет ничего плохого, особенно если они содержат программный код только на языке Python. На платформах, где отсутствует функция putenv, для настройки окружения порождаемой программы можно передавать словарь os.environ инструментам запуска программ в виде параметра.

Стандартные потоки ввода-вывода

Модуль sys – это место обитания стандартных потоков ввода, вывода и вывода ошибок для программ на языке Python. Эти потоки обеспечивают еще один распространенный способ организации обмена данными между программами:

```
>>> import sys
>>> for f in (sys.stdin, sys.stdout, sys.stderr): print(f)
...
<_io.TextIOWrapper name='<stdin>' encoding='cp437'>
<_io.TextIOWrapper name='<stdout>' encoding='cp437'>
<_io.TextIOWrapper name='<stderr>' encoding='cp437'>
```

Стандартные потоки – это всего лишь предварительно открытые объекты файлов Python, которые автоматически подключаются к стандартным потокам ввода-вывода программы при запуске. По умолчанию все они связаны с окном консоли, в котором был запущен интерпретатор Python (или программа на языке Python). Поскольку встроенные функции print и input являются не чем иным, как дружественными интерфейсами к стандартным потокам вывода-ввода, по своему действию 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
19

>>> 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: при запуске программ на языке Python из проводника Windows щелчком на имени файла с расширением *.py* (или с помощью `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 текстом программы расширение *.pyw*, а не *.py*. Расширение *.pyw* означает просто исходный файл *.py* программы, для запуска которой не требуется открывать окно DOS в Windows (это обеспечивается настройками в реестре Windows, где файлам с расширением *.pyw* поставлена в соответствие специализированная версия Python). Файлы с расширением *.pyw* могут импортироваться, как обычные файлы *.py*.

Обратите также внимание, что при запуске программы щелчком мыши вывод производится во всплывающее окно DOS, поэтому сценарии, которые просто выводят текст и завершают свою работу, производят странную «вспышку»: при запуске появляется окно консоли DOS, в него производится вывод, а затем окно сразу закрывается (не самое дружественное поведение!). Чтобы сохранить окно DOS открытым и получить возможность ознакомиться с результатами работы сценария, просто добавьте вызов функции `input()` в конец сценария, который приостановит выполнение до нажатия на клавишу Enter.

Перенаправление потоков ввода-вывода в файлы и программы

Теоретически, текст, который выводится в стандартный поток вывода (и с помощью функции `print`), отображается в окне консоли, в котором запущена программа, текст в стандартный поток ввода (и возвращаемый функцией `input`) поступает с клавиатуры, а стандартный поток вывода ошибок обычно выводит сообщения об ошибках Python в окно консоли. По крайней мере, так происходит по умолчанию. Существует также возможность *перенаправить* эти потоки в файлы или в другие программы системной оболочки, а также в произвольные объекты внутри сценария на языке Python. В большинстве систем возможность перенаправления упрощает повторное использование и комбинирование утилит командной строки общего назначения.

Перенаправление удобно использовать, например, для ввода заранее подготовленных тестовых данных: один и тот же сценарий можно протестировать, используя несколько наборов входных данных, просто перенаправляя стандартный поток ввода при каждом запуске сценария

в разные файлы. Аналогично перенаправление стандартного потока вывода позволяет сохранить и впоследствии проанализировать вывод программы. Например, в системах тестирования для обнаружения ошибок может выполняться сравнение сохраненного стандартного вывода сценария с файлом, содержащим ожидаемые результаты.

Несмотря на всю мощь этой парадигмы, сам механизм перенаправления весьма прост в использовании. В качестве примера рассмотрим простой цикл «прочесть-вычислить-вывести», представленный в примере 3.5.

Пример 3.5. PP4E\System\Streams\teststreams.py

```
"читает числа до символа конца файла и выводит их квадраты"

def interact():
    print('Hello stream world')      # print выводит в sys.stdout
    while True:
        try:
            reply = input('Enter a number>') # input читает из sys.stdin
        except EOFError:
            break                    # исключение при встрече символа 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 автоматически, если файл не импортируется, а выполняется как самостоятельный сценарий. По умолчанию запуск этого файла из командной строки вызывает появление стандартного потока в месте, где вводилась команда. Сценарий просто читает числа, пока не достигнет конца файла в стандартном потоке ввода (в Windows конец файла обычно можно ввести комбинацией двух клавиш Ctrl+Z; в Unix нужно нажать комбинацию Ctrl+D):[1]

[1] Обратите внимание, что функция `input` возбуждает исключение, сигнализируя о конце файла, однако методы чтения файлов просто возвращают в этом случае пустую строку. Поскольку `input` также отсекает символ конца строки, то при чтении пустой строки из файла также возвращается пустая строка, поэтому исключение совершенно необходимо, чтобы определить конец файла. Методы чтения файлов сохраняют символ конца строки и возвращают для пустых строк символ `\n` , а не "". Это одно из отличий прямого чтения из `sys.stdin` от функции `input`. Последняя также принимает в качестве аргумента строку приглашения к вводу, которая автоматически выводится перед приемом входных данных.

```
C:\...\PP4E\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>^Z
Bye
```

И в Windows, и в Unix-подобных системах стандартный поток ввода можно перенаправить в файл – с помощью синтаксической конструкции `< filename` оболочки. Ниже приводится сеанс работы в окне консоли DOS под Windows, где сценарий читает входные данные из текстового файла *input.txt*. То же самое можно проделать и в Linux, только команду DOS `type` нужно заменить командой Unix `cat`:

```
C:\...\PP4E\System\Streams> type input.txt
8
6
C:\...\PP4E\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 ввод данных, которые обычно поступают с клавиатуры в интерактивном режиме, автоматизирован с помощью файла *input.txt*: сценарий читает данные из этого файла, а не с клавиатуры. Точно так же можно перенаправить в файл и стандартный поток вывода – с помощью синтаксической конструкции `> filename` оболочки. При этом перенаправление ввода и вывода можно объединить в одной команде:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt > output.txt

C:\...\PP4E\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 стандартные потоки ввода и вывода сценария отображаются в текстовые файлы, а не в сеанс интерактивной консоли.

Соединение программ с помощью каналов

В Windows и в Unix-подобных системах имеется возможность направлять стандартный вывод одной программы в стандартный ввод другой, помещая между командами символ `|`. Обычно это называется операцией создания «канала» или «конвейера»: оболочка создает канал, соединяющий вывод и ввод двух команд. Попробуем отправить вывод сценария на вход программы `more`, чтобы увидеть, как действует этот прием:

```
C:\...\PP4E\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
```

В этом примере данные также поступают в поток стандартного ввода сценария `teststreams` из файла, но выходные данные (которые выводятся вызовами функции `print`) посылаются другой программе, а не в файл или окно. Принимающей программой является `more` – стандартная программа командной строки для постраничного просмотра, имеющаяся в Windows и в Unix-подобных системах. Поскольку Python привязывает сценарии к стандартной модели потоков ввода-вывода, сценарии на языке Python можно использовать с обоих концов канала: вывод одного сценария Python всегда можно отправить на ввод другого:

```
C:\...\PP4E\System\Streams> type writer.py
print("Help! Help! I'm being repressed!")
print(42)

C:\...\PP4E\System\Streams> type reader.py
print('Got this: "%s"' % input())
import sys
data = sys.stdin.readline()[:-1]
print('The meaning of life is', data, int(data) * 2)

C:\...\PP4E\System\Streams> python writer.py
Help! Help! I'm being repressed!
42

C:\...\PP4E\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 связь устанавливается между двумя программами на языке Python. Сценарий `reader` получает входные данные от сценария `writer` – оба сценария просто используют стандартные функции чтения и записи, не задумываясь о работе механизма потоков. На практике такое соединение программ в цепочку является простой формой организации взаимодействий между программами. Оно облегчает *повторное использование* утилит, предусматривающих возможность взаимодействий через `stdin` и `stdout`, самыми неожиданными способами. Например, программу на языке Python, которая сортирует текст, поступающий из `stdin`, можно использовать для работы с любым источником данных, в том числе с выводом других сценариев. Рассмотрим сценарии командной строки из примеров 3.6 и 3.7, которые сортируют строки с числами, поступающие в стандартный поток ввода, и складывают их.

Пример 3.6. PP4E\System\Streams\sorter.py

```
import sys                              # или sorted(sys.stdin)
lines = sys.stdin.readlines()           # читает входные строки из stdin,
lines.sort()                            # сортирует их
for line in lines: print(line, end='') # отправляет результаты в stdout
                                        # для дальнейшей обработки
```

Пример 3.7. PP4E\System\Streams\adder.py

```
import sys
sum = 0
while True:
    try:
        line = input()     # или sys.stdin.readlines()
    except EOFError:       # или for line in sys.stdin:
        break              # input отсекает символы \n в конце строк
    else:
        sum += int(line)   # во 2-м издании использовалась функция sting.atoi()
print(sum)
```

Мы можем использовать эти универсальные инструменты командной строки, чтобы с их помощью сортировать и складывать содержимое произвольных файлов и вывода других программ (примечание для пользователей Windows: на моей предыдущей машине с Windows XP и Python 2.X я должен был вводить команду «python file.py», а не просто «file.py», в противном случае перенаправление не давало ожидаемых результатов; ныне, в Windows 7 и Python 3.X, обе формы команд действуют корректно):

```
C:\...\PP4E\System\Streams> type data.txt
123
000
999
042

C:\...\PP4E\System\Streams> python sorter.py < data.txt      сортировка файла
000
042
123
999

C:\...\PP4E\System\Streams> python adder.py < data.txt       вычисление суммы
1164

C:\...\PP4E\System\Streams> type data.txt | python adder.py  вычисление суммы
1164                                                         для вывода
                                                             команды type
C:\...\PP4E\System\Streams> type writer2.py
for data in (123, 0, 999, 42):
    print('%03d' % data)
```

```
C:\...\PP4E\System\Streams> python writer2.py | python sorter.py сортировка
000                                                              вывода сценария
042
123
999

C:\...\PP4E\System\Streams> writer2.py | sorter.py      краткая форма записи
        выводит те же результаты, что и предыдущая команда Windows...

C:\...\PP4E\System\Streams> python writer2.py | python sorter.py | python adder.
py
1164
```

В последней команде составлена цепочка из трех сценариев на языке Python: вывод каждого предыдущего сценария соединяется с вводом последующего с помощью синтаксиса конвейера.

Альтернативные реализации сценариев adder и sorter

Если присмотреться, можно заметить, что сценарий *sorter.py* читает сразу все данные, имеющиеся в `stdin`, используя метод `readlines`, а сценарий *adder.py* читает данные по одной строке. Если источником входных данных является другая программа, то в некоторых системах соединенные каналом программы выполняются *параллельно*. В таких системах, особенно если пересылается большой объем данных, лучше производить построчное чтение: читающей программе не придется ждать, пока пишущая программа полностью завершит работу, чтобы заняться обработкой данных. Так как функция `input` просто читает данные из потока `stdin`, схему построчного ввода, используемую в *adder.py*, можно также реализовать прямым обращением к `sys.stdin`:

```
C:\...\PP4E\System\Streams> type adder2.py
import sys
sum = 0
while True:
    line = sys.stdin.readline()
    if not line: break
    sum += int(line)
print(sum)
```

Данная версия использует тот факт, что функция `int` допускает наличие пробельных символов вокруг числа (функция `readline` возвращает строку вместе с символом `\n`, но мы не должны использовать `[:-1]` или `rstrip()` для его удаления). Фактически для достижения того же эффекта можно использовать более современные итераторы файлов – цикл `for`, например, автоматически извлекает из объекта файла по одной строке в каждой итерации (подробнее об итераторах файлов рассказывается в следующей главе):

```
C:\...\PP4E\System\Streams> type adder3.py
import sys
```

```
sum = 0
for line in sys.stdin: sum += int(line)
print(sum)
```

Однако перевод сценария `sorter` на построчное чтение едва ли даст большой выигрыш в производительности, потому что метод `sort` списков требует, чтобы весь список был заполнен. Как будет показано в главе 18, запрограммированные вручную алгоритмы сортировки, скорее всего, будут работать значительно медленнее, чем метод сортировки списка Python.

Интересно отметить, что в версии Python 2.4 и выше эти два сценария можно реализовать более компактно, используя новую встроенную функцию `sorted`, выражения-генераторы и итераторы файлов. Следующий сценарий действует точно так же, как и оригиналы, но имеет заметно меньший размер:

```
C:\...\PP4E\System\Streams> type sorterSmall.py
import sys
for line in sorted(sys.stdin): print(line, end='')

C:\...\PP4E\System\Streams> type adderSmall.py
import sys
print(sum(int(line) for line in sys.stdin))
```

В последнем примере функции `sum` в виде аргумента передается выражение-генератор, по своему поведению похожее на генератор списков, с той лишь разницей, что оно возвращает результаты по одному значению, а не в виде списка. В результате сценарий получается более компактным. За дополнительной информацией обращайтесь к ресурсам по основам языка, таким как книга «Изучаем Python».

Перенаправление потоков и взаимодействие с пользователем

Выше в этом разделе мы направили вывод сценария *teststreams.py* на вход стандартной программы командной строки `more` с помощью следующей команды:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt | more
```

Но поскольку в предыдущей главе мы уже написали на языке Python собственную утилиту «more» постраничного вывода, почему не сделать так, чтобы она тоже принимала ввод из `stdin`? Например, если изменить последние три строки в файле *more.py*, представленном в примере 2.1, на следующие:

```
if __name__ == '__main__':      # если выполняется, а не импортируется
    import sys
    if len(sys.argv) == 1:      # вывести данные из stdin, если нет аргументов
        more(sys.stdin.read())
```

```
else:
    more(open(sys.argv[1]).read())
```

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

```
C:\...\PP4E\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 можно использовать в сценариях на языке Python. Здесь сценарий *teststreams.py* снова принимает данные из файла. И, как и в предыдущем разделе, вывод одной программы отправляется по каналу на ввод другой – сценарий *more.py* в родительском (`..`) каталоге.

Однако в предыдущей команде `more.py` кроется малозаметная проблема. В действительности эта цепочка сработала по чистой случайности: если первый сценарий будет выводить слишком много данных и сценарию `more` придется спрашивать пользователя о разрешении продолжить вывод, произойдет полный отказ сценария (точнее, когда функция `input` возбудит исключение `EOFError`).

Проблема в том, что улучшенный вариант *more.py* использует `stdin` для достижения двух различных целей. Он читает ответ пользователя из `stdin`, вызывая `input`, но теперь *еще* и принимает из `stdin` основные входные данные. Когда поток `stdin` подключается к входному файлу или каналу, его нельзя использовать для получения ответа от пользователя, потому что он содержит входные данные. Кроме того, поскольку перенаправление потока `stdin` происходит еще до запуска программы, невозможно узнать, был ли он перенаправлен в командной строке.

Если потребуется принимать входные данные из `stdin` *и* использовать консоль для взаимодействия с пользователем, в сценарий нужно будет внести дополнительные изменения: нам придется отказаться от использования функции `input` и задействовать специальные интерфейсы для чтения ответов пользователя непосредственно с клавиатуры. В Windows такую возможность обеспечивает модуль `msvcrt`, входящий в состав стандартной библиотеки Python; в большинстве Unix-подобных систем достаточно будет использовать файл устройства */dev/tty*.

Поскольку это достаточно редкая ситуация, мы рассмотрим полную реализацию предложенного решения. В примере 3.8 показана модифицированная версия сценария *more.py* для Windows, которая постранично выводит данные из стандартного потока ввода при вызове без аргументов, а кроме того, в ней используются низкоуровневые и зависящие от платформы средства взаимодействия с пользователем через клавиатуру.

Пример 3.8. PP4E\System\Streams\moreplus.py

```
"""
обеспечивает постраничный вывод в stdout содержимого строки, файла или потока;
если запускается как самостоятельный сценарий, обеспечивает постраничный вывод
содержимого потока stdin или файла, имя которого указывается в виде аргумента
командной строки; когда входные данные поступают через поток stdin, исключается
возможность использовать его для получения ответов пользователя --вместо этого
можно использовать платформозависимые инструменты или графический интерфейс;
"""

import sys

def getreply():
    """
    читает клавишу, нажатую пользователем,
    даже если stdin перенаправлен в файл или канал
    """
    if sys.stdin.isatty():              # если stdin связан с консолью,
        return input('?')               # читать ответ из stdin
    else:
        if sys.platform[:3] == 'win':   # если stdin был перенаправлен,
            import msvcrt               # его нельзя использовать для чтения
            msvcrt.putch(b'?')          # ответа пользователя
            key = msvcrt.getche()       # использовать инструмент консоли
            msvcrt.putch(b'\n')         # getch(), которая не выводит символ
            return key                  # для нажатой клавиши
        else:
            assert False, 'platform not supported'
            #для Linux: open('/dev/tty').readline()[:-1]

def more(text, numlines=10):
    """
    реализует постраничный вывод содержимого строки в stdout
    """
    lines = text.splitlines()
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print(line)
        if lines and getreply() not in [b'y', b'Y']: break

if __name__ == '__main__':              # если выполняется, а не импортируется
    if len(sys.argv) == 1:              # если нет аргументов командной строки
        more(sys.stdin.read())          # вывести содержимое stdin
    else:
        more(open(sys.argv[1]).read()) # иначе вывести содержимое файла
```

Большая часть нововведений этой версии находится в функции `getreply`**. Метод файла** `isatty` **сообщает, соединен ли** `stdin` **с консолью, – если да, функция просто считывает ответ из** `stdin`**, как и раньше. Конечно, по-**

добная дополнительная логика необходима только в сценариях, предусматривающих возможность взаимодействия с пользователем *и* получения входных данных из stdin. В приложениях с графическим интерфейсом можно было бы, например, выводить диалог, реализовать обработку событий от клавиатуры в виде функций обратного вызова и так далее (знакомиться с графическими интерфейсами мы будем в главе 7).

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('adderSmall.py').read())
import sys
print(sum(int(line) for line in sys.stdin))
```

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

```
C:\...\PP4E\System\Streams> python moreplus.py adderSmall.py
import sys
print(sum(int(line) for line in sys.stdin))

C:\...\PP4E\System\Streams> python moreplus.py moreplus.py
"""
обеспечивает постраничный вывод в stdout содержимого строки, файла или потока;
если запускается как самостоятельный сценарий, обеспечивает постраничный вывод
содержимого потока stdin или файла, имя которого указывается в виде аргумента
командной строки; когда входные данные поступают через поток stdin, исключается
возможность использовать его для получения ответов пользователя - вместо этого
можно использовать платформозависимые инструменты или графический интерфейс;
"""

import sys

def getreply():
?n
```

Но теперь сценарий также правильно выводит постранично текст, перенаправленный в stdin из *файла* или по *конвейеру* команд, даже если этот текст слишком велик, чтобы целиком уместиться в одну страницу. В большинстве оболочек входные данные посылаются с помощью перенаправления или операторов конвейера, как показано ниже:

```
C:\...\PP4E\System\Streams> python moreplus.py < moreplus.py
"""
обеспечивает постраничный вывод в stdout содержимого строки, файла или потока;
если запускается как самостоятельный сценарий, обеспечивает постраничный вывод
```

```
содержимого потока stdin или файла, имя которого указывается в виде аргумента
командной строки; когда входные данные поступают через поток stdin, исключается
возможность использовать его для получения ответов пользователя - вместо этого
можно использовать платформозависимые инструменты или графический интерфейс;
"""

import sys

def getreply():
?n

C:\...\PP4E\System\Streams> type moreplus.py | python moreplus.py
"""
обеспечивает постраничный вывод в stdout содержимого строки, файла или потока;
если запускается как самостоятельный сценарий, обеспечивает постраничный вывод
содержимого потока stdin или файла, имя которого указывается в виде аргумента
командной строки; когда входные данные поступают через поток stdin, исключается
возможность использовать его для получения ответов пользователя - вместо этого
можно использовать платформозависимые инструменты или графический интерфейс;
"""

import sys

def getreply():
?n
```

Наконец, если вывод одного сценария отправляется по каналу на ввод другого, все работает как надо, и при этом взаимодействие с пользователем не вызывает нарушений (и вовсе не потому, что нам просто повезло):

```
......\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 подается на стандартный *ввод* другого сценария, находящегося в том же каталоге: *moreplus.py* читает вывод *teststreams.py*.

Все перенаправления в таких командах действуют только потому, что сценариям безразлично, чем в действительности являются стандартный ввод и вывод – консолью, файлами или каналами между программами. Например, при запуске *moreplus.py* как самостоятельного сценария он просто читает поток `sys.stdin`; командная оболочка (например, DOS в Windows, csh в Linux) прикрепляет такие потоки к источникам, определяемым командой, перед запуском сценария. Для доступа к этим источникам сценарии используют заранее открытые объекты файлов `stdin` и `stdout`, независимо от их истинной природы.

Для читателей, ведущих подсчет: мы запускали один сценарий постраничного вывода `more` четырьмя различными способами – импортируя и вызывая его функцию, передавая имя файла через аргумент командной строки, перенаправляя `stdin` в файл и передавая вывод команды по каналу в `stdin`. Благодаря возможности импортировать функции, принимать аргументы командной строки и получать ввод через стандартные потоки системные инструменты Python можно повторно использовать в разнообразных режимах.

Перенаправление потоков в объекты Python

Все приведенные выше способы перенаправления стандартных потоков действуют для программ, написанных на любом языке программирования, который обеспечивает возможность перехватывать стандартные потоки, и зависят скорее от процессора командной строки оболочки, чем от самого интерпретатора. Операции перенаправления в командной строке, такие как `< filename` и `| program`, обрабатываются оболочкой, а не интерпретатором Python. Более «питонистый» способ перенаправления можно реализовать в самих сценариях, присваивая переменным `sys.stdin` и `sys.stdout` объекты, похожие на файлы.

Это возможно благодаря тому, что любой объект, который набором методов напоминает файл, может выступать в роли стандартного потока. Важен не конкретный тип объекта, а его интерфейс (иногда его называют протоколом). Это означает следующее:

- Любой объект, обладающий методами *чтения*, может быть присвоен переменной `sys.stdin`, в результате чего ввод будет осуществляться через методы чтения этого объекта.
- Любой объект, обладающий методами *записи*, может быть присвоен переменной `sys.stdout`; в результате весь стандартный вывод будет отправляться методам этого объекта.

Так как функции `print` и `input` просто вызывают методы `write` и `readline` объектов, на которые ссылаются `sys.stdout` и `sys.stdin`, мы можем генерировать и перехватывать стандартные текстовые потоки с помощью объектов, реализованных с помощью классов.

Если вам уже приходилось изучать язык Python, вы, вероятно, знаете, что подобная совместимость по интерфейсам обычно называется *полиморфизмом* – неважно, чем является объект, и неважно, что делают его методы, – важно, чтобы этот объект предоставлял ожидаемый интерфейс. Такое либеральное отношение к типам данных объясняет значительную долю гибкости и выразительности программного кода на языке Python. Ниже демонстрируется способ, позволяющий сценариям переопределять собственные потоки ввода-вывода. В примере 3.9 приводится вспомогательный модуль, демонстрирующий эту идею.

Пример 3.9. PP4E\System\Streams\redirect.py

```
"""
объекты, похожие на файлы, один из которых сохраняет в строке текст,
отправленный в стандартный поток вывода, а другой обеспечивает ввод текста
из строки в стандартный поток ввода; функция redirect вызывает переданную
ей функцию, для которой стандартные потоки вывода и ввода будут связаны
с объектами, похожими на файлы;
"""
import sys                                  # импортировать встроенный модуль

class Output:                               # имитирует выходной файл
    def __init__(self):
        self.text = ''                      # при создании строка пустая
    def write(self, string):                # добавляет строку байтов
        self.text += string
    def writelines(self, lines):            # добавляет все строки в список
        for line in lines: self.write(line)

class Input:                                # имитирует входной файл
    def __init__(self, input=''):           # аргумент по умолчанию
        self.text = input                   # сохранить строку при создании
    def read(self, size=None):              # необязательный аргумент
        if size == None:                    # прочитать N байт или все
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:size], self.text[size:]
        return res
    def readline(self):
        eoln = self.text.find('\n')         # найти смещение следующего eoln
        if eoln == -1:                      # извлечь строку до eoln
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:eoln+1], self.text[eoln+1:]
        return res

def redirect(function, pargs, kargs, input): # перенаправляет stdin/out
    savestreams = sys.stdin, sys.stdout     # вызывает объект функции
    sys.stdin = Input(input)                # возвращает текст в stdout
    sys.stdout = Output()
    try:
        result = function(*pargs, **kargs)  # вызвать функцию с аргументами
        output = sys.stdout.text
    finally:                                # восстановить, независимо от
        sys.stdin, sys.stdout = savestreams # того, было ли исключение
    return (result, output)                 # вернуть результат,
                                            # если исключения не было
```

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

`Output`

Предоставляет интерфейс (он же протокол) метода записи, предполагаемый у выходных файлов, но сохраняет всю записываемую информацию в строке, хранящейся в памяти.

`Input`

Предоставляет интерфейс, предполагаемый у входных файлов, но возвращает входные данные по требованию, извлекая их из хранящейся в памяти строки, переданной при создании объекта.

Функция `redirect` в конце этого файла объединяет эти два объекта, чтобы выполнить единственную функцию, для которой стандартные потоки ввода и вывода будут перенаправлены в объекты Python. Функции, которая вызывается функцией `redirect`, не требуется ни знать, ни заботиться о том, что вызываемые ею функции `print` и `input` или методы `stdin` и `stdout` в действительности будут иметь дело с нашими объектами, а не с настоящим файлом, каналом или пользователем.

Чтобы продемонстрировать, как действует эта функция, импортируем и вызовем функцию `interact`, лежащую в основе сценария `teststreams`, представленного в примере 3.5, который прежде мы запускали из командной строки (для использования вспомогательной функции перенаправления нужно действовать на языке функций, а не файлов). При непосредственном вызове функция читает данные с клавиатуры и выводит результаты на экран, как если бы она выполнялась как программа без перенаправления:

```
C:\...\PP4E\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^Z
Bye
>>>
```

Теперь вызовем эту функцию под управлением функции перенаправления в *redirect.py* и передадим ей некоторый готовый входной текст. В этом случае на вход функции `interact` поступит переданная строка (`'4\n5\n6\n'` – три строки с явными символами конца строки), а результатом выполнения функции будет кортеж, содержащий возвращаемое значение и строку с текстом, который был записан в стандартный поток вывода:

```
>>> from redirect import redirect
>>> (result, output) = redirect(interact, (), {}, '4\n5\n6\n')
```

```
>>> print(result)
None
>>> output
'Hello stream world\nEnter a number>4 squared is 16\nEnter a number>5 squared
is 25\nEnter a number>6 squared is 36\nEnter a number>Bye\n'
```

На выходе получится одна длинная строка, содержащая весь текст, записанный в стандартный поток вывода. Чтобы улучшить внешний вид строки, ее можно передать функции `print` или разбить на отдельные строки с помощью строкового метода `splitlines`:

```
>>> for line in output.splitlines(): 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 написали в предыдущей главе (пример 2.1). При этом придется меньше запоминать и вводить с клавиатуры, а качество работы уже проверено нами (ниже, как и во всех примерах, где выполняется импортирование модулей из других каталогов, предполагается, что каталог, содержащий корневой подкаталог *PP4E*, находится в пути поиска модулей, – измените значение переменной окружения `PYTHONPATH`, если это необходимо):

```
>>> from PP4E.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 продемонстрированные приемы могут иметь широкое применение. Например, в программу, реализующую интерфейс командной строки для взаимодействия с пользователем, легко добавить графический интерфейс. Нужно просто перехватить стандартный вывод с помощью объекта, такого как экземпляр класса `Output`, и сбросить текстовую строку в окно. Аналогично стандартный ввод можно перенаправить в объект, который получает текст из графического интерфейса (например, из окна диалога). Поскольку классы могут заменять настоящие файлы, их можно использовать в любых инструментах, работающих с файлами. Обратитесь к модулю перенаправления потоков с графическим интерфейсом `guiStreams` в главе 10, где вы найдете конкретную реализацию некоторых из описанных идей.

Вспомогательные классы io.StringIO и io.BytesIO

Прием перенаправления потоков ввода-вывода в объекты, о котором рассказывалось в предыдущем разделе, оказался настолько удобным на практике, что в стандартную библиотеку был добавлен модуль, автоматизирующий задачу его применения в большинстве случаев (хотя в некоторых случаях, например, при реализации графического интерфейса, все еще может потребоваться писать свой программный код). В стандартной библиотеке имеется объект, который добавляет интерфейс объектов файлов к объектам строк, хранящимся в памяти. Например:

```
>>> from io import StringIO
>>> buff = StringIO()              # сохраняет записываемый текст в строке
>>> buff.write('spam\n')
5
>>> buff.write('eggs\n')
5
>>> buff.getvalue()
'spam\neggs\n'

>>> buff = StringIO('ham\nspam\n') # возвращает входные данные из строки
>>> buff.readline()
'ham\n'
>>> buff.readline()
'spam\n'
>>> buff.readline()
''
```

Экземпляры класса `StringIO` могут присваиваться переменным `sys.stdin` и `sys.stdout`, как демонстрировалось в предыдущем разделе, с целью перенаправить потоки для функций `input` и `print`, и использоваться в любом программном коде, выполняющем операции с настоящими объектами файлов. Напомню еще раз, что в языке Python правила игры определяются интерфейсом объекта, а не его конкретным типом:

```
>>> from io import StringIO
>>> import sys
>>> buff = StringIO()

>>> temp = sys.stdout
>>> sys.stdout = buff
>>> print(42, 'spam', 3.141)   # или print(..., file=buff)
>>> sys.stdout = temp          # восстановит оригинальный поток
>>> buff.getvalue()
'42 spam 3.141\n'
```

Следует также отметить, что существует класс `io.BytesIO`, обладающий похожим поведением, но он отображает операции с файлами не на строку типа `str`, а на буфер байтов типа `bytes`:

```
>>> from io import BytesIO
>>> stream = BytesIO()
>>> stream.write(b'spam')
>>> stream.getvalue()
b'spam'

>>> stream = BytesIO(b'dpam')
>>> stream.read()
b'dpam'
```

Из-за существенных различий между текстовыми и двоичными данными в Python 3.X эта альтернатива может оказаться более подходящей для сценариев, имеющих дело с двоичными данными. Подробнее с проблемой различий между текстовыми и двоичными данными мы познакомимся в следующей главе, когда займемся исследованием файлов.

Перехват потока stderr

Мы сосредоточились на перенаправлении `stdin` и `stdout`, но поток `stderr` также можно перенаправлять в файлы, каналы и объекты. Несмотря на то, что некоторые оболочки поддерживают возможность перенаправления этого потока, тем не менее это также можно сделать легко и просто в сценарии Python. Например, присвоение переменной `sys.stderr` экземпляра класса, такого как `Output` или `StringIO` из предыдущего примера, позволит сценарию перехватывать также текст, записываемый в стандартный поток ошибок.

Сам интерпретатор Python использует стандартный поток ошибок для вывода сообщений об ошибках (графический интерфейс IDLE перехватывает этот текст и по умолчанию окрашивает его в красный цвет). Однако в языке отсутствуют высокоуровневые инструменты для работы со стандартным потоком ошибок, такие как функции `print` и `input` для стандартных потоков вывода и ввода. Если вам потребуется организовать вывод в стандартный поток ошибок, вы можете явно вызвать метод `sys.stderr.write()` или прочитать следующий раздел, где описывается одна особенность функции `print`, упрощающая эту возможность.

Операция перенаправления стандартного потока ошибок из командной строки выглядит несколько сложнее и хуже переносится. В большинстве Unix-подобных систем перехватить вывод в поток `stderr` обычно можно с помощью операции перенаправления вида `command > output 2>&1`. Однако в некоторых версиях Windows она не действует, и даже в некоторых оболочках для Unix она может иметь другой вид – за дополнительной информацией обращайтесь к страницам справочного руководства по вашей оболочке.

Возможность перенаправления с помощью функции print

Вследствие того, что переназначение атрибутов потоков приобрело большую популярность, встроенная функция `print` в языке Python также

была дополнена возможностью явно указывать файл для вывода. Следующая инструкция:

```
print(stuff, file=afile) # afile - это объект, а не имя строковой переменной
```

выведет `stuff` в `afile`, а не в поток `sys.stdout`. По своему действию это напоминает присваивание объекта переменной `sys.stdout`, но в данном случае отпадает необходимость сохранять и восстанавливать первоначальное значение, чтобы вернуться к использованию оригинального потока вывода (как было показано в разделе, описывающем перенаправление потоков в объекты). Например:

```
import sys
print('spam' * 2, file=sys.stderr)
```

выведет текст в объект стандартного потока ошибок, а не в `sys.stdout`, причем такое перенаправление будет действовать только для данного вызова функции `print`. Следующий вызов функции `print` (без аргумента `file`) выведет текст в стандартный поток вывода, как обычно. Точно так же в качестве выходного файла можно передать свой собственный объект или экземпляр класса из стандартной библиотеки:

```
>>> from io import StringIO
>>> buff = StringIO()
>>> print(42, file=buff)
>>> print('spam', file=buff)
>>> print(buff.getvalue())
42
spam

>>> from redirect import Output
>>> buff = Output()
>>> print(43, file=buff)
>>> print('eggs', file=buff)
>>> print(buff.text)
43
eggs
```

Другие варианты перенаправления: еще раз об os.popen и subprocess

Ближе к концу предыдущей главы мы впервые встретились с функцией `os.popen` и родственной ей `subprocess.Popen`, которые предоставляют возможность перенаправления потоков ввода-вывода других команд из программы на языке Python. Как мы видели, эти инструменты могут использоваться для выполнения команд оболочки (например, команд, которые обычно вводятся с клавиатуры в ответ на приглашение DOS или csh), и они возвращают объект Python, похожий на файл, соединенный с потоком вывода команды, – чтение из объекта файла позволяет

сценарию принимать вывод другой программы. Однако эти инструменты могут также использоваться для соединения с потоками ввода.

Благодаря этому функцию os.popen и инструменты из модуля subprocess можно рассматривать как еще один способ перенаправления потоков порождаемых программ, родственный только что рассмотренным приемам. Их действие во многом похоже на действие оператора | объединения команд в конвейер (фактически имена этих инструментов означают «pipe open» – «открыть канал»), но они выполняются внутри сценария и предоставляют схожий с файлами интерфейс к потокам данных, связанных каналом. По духу они близки функции redirect, но запускают не функции, а программы, и потоки ввода-вывода обрабатываются в порождающем сценарии как файлы (не привязанные к объектам классов). Эти инструменты перенаправляют потоки ввода-вывода программ, запускаемых сценарием, а не самого сценария.

Перенаправление ввода или вывода с помощью os.popen

Передавая в функцию флаг нужного режима, мы фактически выполняем перенаправление в файл потока ввода *или* вывода программы, порожденной сценарием, и можем получить код завершения этой программы вызовом метода close (значение None говорит об успешном завершении). Чтобы проиллюстрировать это, рассмотрим следующие два сценария:

```
C:\...\PP4E\System\Streams> type hello-out.py
print('Hello shell world')

C:\...\PP4E\System\Streams> type hello-in.py
inp = input()
open('hello-in.txt', 'w').write('Hello ' + inp + '\n')
```

Эти сценарии могут запускаться из командной строки, как обычно:

```
C:\...\PP4E\System\Streams> python hello-out.py
Hello shell world

C:\...\PP4E\System\Streams> python hello-in.py
Brian

C:\...\PP4E\System\Streams> type hello-in.txt
Hello Brian
```

В предыдущей главе мы видели, что сценарии на языке Python могут также читать *вывод* других программ и подобных им сценариев, как показано ниже:

```
C:\...\PP4E\System\Streams> python
>>> import os
>>> pipe = os.popen('python hello-out.py')   # 'r' - по умолчанию, чтение stdout
```

```
>>> pipe.read()
'Hello shell world\n'
>>> print(pipe.close())                    # код завершения: None - успех
None
```

Но сценарии на языке Python могут играть роль *источников данных*, подаваемых в поток стандартного ввода порождаемых программ, – если передать функции os.popen флаг режима «w» вместо «r», подразумеваемого по умолчанию, она вернет объект, подключенный к потоку ввода порожденной программы. Все, что мы запишем в этот объект со стороны родительского сценария, окажется в стандартном потоке ввода запущенной программы:

```
>>> pipe = os.popen('python hello-in.py', 'w')   # 'w'- запись в stdin программы
>>> pipe.write('Gumby\n')
6
>>> pipe.close()                                 # символ \n в конце необязателен
>>> open('hello-in.txt').read()                  # вывод был отправлен в файл
'Hello Gumby\n'
```

Функция popen достаточно сообразительна, чтобы выполнить команду оболочки как независимый процесс на платформах, поддерживающих такую возможность. Она принимает необязательный третий параметр, который используется для управления буферизацией записываемого текста, но мы пока не будем затрагивать эту тему.

Перенаправление ввода и вывода с помощью модуля subprocess

Еще больший контроль над потоками ввода-вывода порождаемых программ позволяет получить модуль subprocess, представленный в предыдущей главе. Как было показано ранее, с помощью этого модуля можно имитировать функциональность os.popen, но он позволяет добиться большего, например – создавать двунаправленные потоки обмена данными (то есть подключаться к обоим потокам, ввода и вывода, порождаемой программы) и связывать вывод одной программы с вводом другой.

Например, этот модуль позволяет множеством способов запускать программу, подключаться к ее стандартному потоку вывода и получать код завершения. Ниже демонстрируются три наиболее типичных способа использования этого модуля для запуска программы и перенаправления ее потока вывода (напомню, что для опробования примеров из этого раздела в Unix-подобных системах вам может потребоваться передавать функции Popen аргумент shell=True, как отмечалось в главе 2):

```
C:\...\PP4E\System\Streams> python
>>> from subprocess import Popen, PIPE, call
>>> X = call('python hello-out.py')           # удобно
Hello shell world
```

```
>>> X
0

>>> pipe = Popen('python hello-out.py', stdout=PIPE)
>>> pipe.communicate()[0]                    # (stdout, stderr)
b'Hello shell world\r\n'
>>> pipe.returncode                          # код завершения
0

>>> pipe = Popen('python hello-out.py', stdout=PIPE)
>>> pipe.stdout.read()
b'Hello shell world\r\n'
>>> pipe.wait()                              # код завершения
0
```

Функция `call`, использованная в первом из этих трех способов, – это всего лишь функция-обертка, реализованная для удобства (существует несколько таких функций, о которых вы сможете прочитать в руководстве по библиотеке языка Python). Функция `communicate` делает второй способ немного удобнее третьего (она позволяет отправлять данные в `stdin`; читать данные из `stdout`, пока не будет достигнут конец файла; и ожидает завершения дочернего процесса).

Перенаправление и подключение к потоку ввода порождаемой программы реализуется так же просто, хотя и немного сложнее, чем при использовании функции `os.popen` с флагом режима `'w'`, как было показано в предыдущем разделе (как уже упоминалось в предыдущей главе, в настоящее время функция `os.popen` реализована с применением инструментов из модуля `subprocess`, и поэтому сама может считаться функцией-оберткой, реализованной для удобства):

```
>>> pipe = Popen('python hello-in.py', stdin=PIPE)
>>> pipe.stdin.write(b'Pokey\n')
6
>>> pipe.stdin.close()
>>> pipe.wait()
0
>>> open('hello-in.txt').read()         # вывод был отправлен в файл
'Hello Pokey\n'
```

С помощью этого модуля мы можем получить доступ *к обоим потокам, ввода и вывода*, порожденной программы. Для демонстрации воспользуемся простыми сценариями, выполняющими операции чтения и записи, созданными нами ранее:

```
C:\...\PP4E\System\Streams> type writer.py
print("Help! Help! I'm being repressed!")
print(42)

C:\...\PP4E\System\Streams> type reader.py
print('Got this: "%s"' % input())
```

```
import sys
data = sys.stdin.readline()[:-1]
print('The meaning of life is', data, int(data) * 2)
```

Следующий программный код демонстрирует возможность чтения и записи в потоки ввода-вывода сценария `reader` – объект `pipe` имеет два атрибута с объектами, похожими на файлы, один из которых подключается к потоку ввода, а другой – к потоку вывода (пользователи Python 2.X легко могут узнать в них эквивалент кортежа, возвращаемого функцией `os.popen2`, ныне исключенной из библиотеки):

```
>>> pipe = Popen('python reader.py', stdin=PIPE, stdout=PIPE)
>>> pipe.stdin.write(b'Lumberjack\n')
11
>>> pipe.stdin.write(b'12\n')
3
>>> pipe.stdin.close()
>>> output = pipe.stdout.read()
>>> pipe.wait()
0
>>> output
b'Got this: "Lumberjack"\r\nThe meaning of life is 12 24\r\n'
```

Как будет показано в главе 5, при двунаправленном обмене данными с программами, подобными этим, необходимо проявлять осторожность – механизм буферизации потоков вывода может приводить к взаимоблокировке при чередовании операций записи и чтения и, в конечном счете, к необходимости использовать для решения проблемы такие инструменты, как `Pexpect` (подробнее об этой функции будет рассказываться далее в книге).

Наконец, модуль `subprocess` позволяет реализовать еще более экзотические формы управления потоками ввода-вывода – ниже демонстрируется соединение *двух программ*, где поток вывода одного сценария на языке Python подключается к потоку ввода другого. Сначала демонстрируется подключение с использованием командной оболочки, а затем – с помощью модуля `subprocess`:

```
C:\...\PP4E\System\Streams> python writer.py | python reader.py
Got this: "Help! Help! I'm being repressed!"
The meaning of life is 42 84

C:\...\PP4E\System\Streams> python
>>> from subprocess import Popen, PIPE
>>> p1 = Popen('python writer.py', stdout=PIPE)
>>> p2 = Popen('python reader.py', stdin=p1.stdout, stdout=PIPE)
>>> output = p2.communicate()[0]
>>> output
b'Got this: "Help! Help! I\'m being repressed!"\r\nThe meaning of life is 42 84\r\n'
```

```
>>> p2.returncode
0
```

Нечто похожее можно реализовать с помощью функции os.popen, но тот факт, что она может подключать только один из потоков ввода-вывода (но не оба), препятствует возможности перехватить вывод второго сценария:

```
>>> import os
>>> p1 = os.popen('python writer.py', 'r')
>>> p2 = os.popen('python reader.py', 'w')
>>> p2.write( p1.read() )
36
>>> X = p2.close()
Got this: "Help! Help! I'm being repressed!"
The meaning of life is 42 84
>>> print(X)
None
```

С точки зрения более широкой перспективы, функция os.popen и модуль subprocess являются переносимыми эквивалентами механизма перенаправления потоков ввода-вывода порождаемых программ, реализованного в командных оболочках для Unix-подобных систем. Однако реализации на языке Python с таким же успехом работают в Windows и предоставляют более универсальный способ запуска других программ из сценариев на языке Python. Строки команд, передаваемые им, могут иметь свои особенности в зависимости от платформы (например, в Unix список содержимого каталога можно получить с помощью команды ls, а в Windows – с помощью команды dir), но сами инструменты могут применяться на всех платформах, поддерживающих Python.

Запуск новых, независимых программ и подключение к их потокам ввода-вывода из родительской программы в Unix-подобных системах можно также реализовать с помощью функций os.fork, os.pipe, os.dup и некоторых функций из семейства os.exec. Кроме того, они обеспечивают еще один способ перенаправления потоков ввода-вывода и являются низкоуровневыми эквивалентами таким инструментам, как os.popen (функция os.fork доступна в Windows, в версии Python для Cygwin).

Однако все перечисленные функции являются более сложными инструментами параллельной обработки данных, поэтому мы отложим дальнейшее их обсуждение до главы 5, где дополнительно будет рассказываться об организации каналов и получении кодов завершения. А к модулю subprocess мы вернемся в главе 6, где на его основе реализуем механизм регрессионного тестирования, перехватывающий все *три* стандартных потока ввода-вывода тестируемого сценария – потоки ввода, вывода и ошибок.

Но перед этим, в главе 4, мы продолжим наше исследование системных интерфейсов, реализованных в библиотеке языка Python, и познако-

мимся с инструментами для работы с файлами и каталогами. Несмотря на то, что все наше внимание будет сконцентрировано на других вопросах, тем не менее мы увидим, что некоторые инструменты, изученные здесь, могут использоваться, как универсальные инструменты системного программирования. Например, возможность запуска команд оболочки позволяет исследовать содержимое каталогов, а интерфейс объектов файлов, на котором мы подробно остановимся в следующей главе, составляет основу приемов работы с потоками ввода-вывода, обсуждавшихся здесь.

Python и csh

Если вы знакомы с другими распространенными языками сценариев командной оболочки, вам может оказаться полезным сравнить их с языком Python. Ниже приводится простой сценарий на языке командной оболочки `csh` для Unix, который отправляет по электронной почте все файлы с расширением *.py* из текущего рабочего каталога (то есть все файлы с исходным программным кодом на языке Python) на фиктивный, как мы надеемся, электронный адрес:

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

Ниже приводится эквивалентный сценарий на языке Python:

```
#!/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))
```

Он выглядит более подробным. Язык Python, в отличие от `csh`, не предназначен для разработки исключительно сценариев командной строки, поэтому системные интерфейсы необходимо импортировать и вызывать явно. А так как Python не является языком программирования, ориентированным на работу исключительно со строками, строки символов необходимо заключать в кавычки, как в языке C.

И хотя использование языка Python для создания простых сценариев, таких как этот, требует лишних нажатий на клавиши, тем не менее его принадлежность к классу языков программирования общего назначения обеспечивает ему более высокую привлекатель-

ность, как только мы покидаем область тривиальных программ. Мы могли бы, к примеру, расширить предыдущий сценарий, добавив в него такие возможности, как передача файлов по протоколу FTP, предоставление выбора операции с помощью графического интерфейса, содержащего строку состояния, извлечение сообщений из базы данных SQL, использование COM-объектов в Windows, – и все это с применением стандартных инструментов Python.

Кроме того, сценарии на языке Python обычно легко могут переноситься на другие платформы, в отличие от `csh`. Например, задействовав для отправки электронной почты модуль Python, обеспечивающий интерфейс к SMTP, вместо утилиты `mail` командной строки, мы сможем использовать этот сценарий на любом компьютере, где установлен Python и имеется подключение к Интернету (как мы узнаем в главе 13, для работы с протоколом SMTP достаточно одних сокетов). Как и в языке C, нам нет необходимости использовать префикс `$`, чтобы получать значения переменных; что еще можно желать от свободного языка?

4

Инструменты для работы с файлами и каталогами

«Как очистить свой жесткий диск за пять простых шагов»

Эта глава продолжает исследование системных интерфейсов Python, фокусируясь на инструментах для работы с файлами и каталогами. Как вы увидите в этой главе, благодаря встроенным инструментам и инструментам из стандартной библиотеки операции с файлами и деревьями каталогов реализуются очень просто. Механизмы работы с файлами составляют часть ядра языка Python, поэтому часть материала этой главы посвящена обзору основных сведений о файлах, подробно рассматриваемых в других книгах, таких как четвертое издание «Изучаем Python», к которым мы рекомендуем обратиться за детальными разъяснениями концепций, связанных с файлами. Например, мы будем касаться таких тем, как итерации, менеджеры контекста и поддержка Юникода объектами файлов, но они не будут раскрываться здесь полностью. Цель этой главы – дать достаточный объем сведений, чтобы вы могли начать писать полезные сценарии.

Инструменты для работы с файлами

Внешние файлы – это тот объект, ради которого чаще всего разрабатываются системные утилиты. Например, система тестирования может читать входные данные из одного файла, сохранять результаты программы в другом файле и проверять полученные результаты, загружая третий файл. Даже программы с пользовательским интерфейсом и веб-приложения могут загружать двоичные изображения и аудиоклипы из локальных файлов. Это базовая концепция программирования.

В Python главным инструментом, используемым сценариями для доступа к файлам, служит встроенная функция `open`. Поскольку эта функция является неотъемлемой частью языка Python, вы, вероятно, уже знакомы с тем, как она работает. При вызове функции `open` возвращается новый *объект файла*, соединенный с внешним файлом. Объект файла обладает методами для чтения и записи данных и для выполнения различных операций над файлами. Кроме того, функция `open` предоставляет *переносимый* интерфейс к используемой файловой системе – она одинаково работает на любой платформе, где выполняется Python.

Другие связанные с файлами модули в Python позволяют, например, выполнять операции над файлами на низком уровне с использованием файловых дескрипторов (модуль `os`), перемещать файлы и группы файлов (модули `os` и `shutil`), сохранять в файлах данные и объекты по ключу (модули `dbm` и `shelve`) и обращаться к базам данных SQL (модуль `sqlite3` и модули сторонних разработчиков). Последние две категории в большей степени относятся к обсуждению баз данных, которое ведется в главе 17.

В данном разделе мы кратко рассмотрим встроенный объект файла и несколько более сложных тем, относящихся к файлам. Как обычно, более подробное описание и методы, которые мы не имеем возможности разместить здесь, следует искать в руководстве по библиотеке или в справочниках, таких как «Python Pocket Reference». Не забывайте, что краткую справку можно получить в интерактивной оболочке: чтобы ознакомиться со списком атрибутов объекта файла, можно вызвать функцию `dir(`*`file`*`)` для объекта открытого файла; вызвав функцию `help(`*`file`*`)`, можно получить справку более общего характера; а с помощью вызова `help(`*`file`*`.read)` – справку о конкретном методе, таком как `read`, хотя реализация объекта файла в версии 3.1 содержит меньше справочной информации, чем руководство по библиотеке и другие ресурсы.

Модель объекта файла в Python 3.X

Как и в случае со строковыми типами, о которых говорилось в главе 2, поддержка файлов в Python 3.X стала гораздо богаче, чем в предыдущих версиях. Как уже отмечалось ранее, в Python 3.X строки типа `str` всегда представляют текст Юникода (символы ASCII или многобайтовые символы), а строки типов `bytes` и `bytearray` представляют простые двоичные данные. Python 3.X проводит подобные различия между файлами, содержащими текст и двоичные данные:

- *Текстовые файлы* содержат текст, состоящий из символов Юникода. Содержимое текстовых файлов в сценариях всегда представляется в виде строк типа `str` – последовательностей символов (точнее, последовательностей «кодовых пунктов» Юникода). Для текстовых файлов автоматически выполняется преобразование символов конца строки, о котором рассказывается в этой главе, а к содержимому

файлов автоматически применяются операции кодирования/декодирования: данные кодируются в двоичное представление при записи в файл и декодируются обратно в Юникод при чтении из файла, в соответствии с указанной или используемой по умолчанию кодировкой. Кодирование является тривиальной операцией для текста ASCII, но может быть весьма сложной в других случаях.

- *Двоичные файлы* содержат обычные 8-битовые байты. Содержимое двоичных файлов в сценариях всегда представляется в виде строк байтов, обычно в виде объекта типа `bytes` – последовательности коротких целых чисел, которые поддерживают большинство операций, присущих типу `str`, и отображаются как последовательности символов ASCII, когда это возможно. Для двоичных файлов не предусматривается никаких преобразований данных при чтении или записи: ни преобразования символов конца строки, ни кодирования/декодирования в Юникод.

На практике текстовые файлы используются для хранения действительно текстовых данных, а двоичные файлы – для хранения таких элементов, как упакованные двоичные данные, изображения, аудиоданные, выполняемый программный код и так далее. Программно эти два типа файлов различаются с помощью аргумента со строкой режима, который передается функции `open`: дополнительный символ «b» (например, `'rb'`, `'wb'`) означает, что файл содержит двоичные данные. Для создания нового содержимого текстовых файлов используются обычные строки (например, `'spam'` или `bytes.decode()`), а для создания нового содержимого двоичных файлов – строки байтов (например, `b'spam'` или `str.encode()`).

Если в вашей практике область применения файлов не ограничивается использованием текста в кодировке ASCII, различия между представлением текстовых и двоичных данных в версии 3.X иногда будут сказываться на вашем программном коде. При работе с текстовыми файлами требуется использовать строки типа `str`, а с двоичными файлами – строки байтов. Поскольку вы не сможете смешивать эти типы в выражениях, вам придется внимательно подходить к вопросу выбора режима открытия файла. Многие встроенные инструменты, которые мы будем использовать в этой книге, делают этот выбор за нас – модули `struct` и `pickle`, например, в версии 3.X работают со строками байтов, а пакет `xml` – с Юникодом. Кроме того, о различиях между текстовыми и двоичными данными в версии 3.X необходимо помнить даже при использовании системных инструментов, таких как дескрипторы каналов и сокеты, потому что на сегодняшний день эти инструменты передают данные в виде строк байтов (впрочем, при необходимости эти данные можно кодировать и декодировать как текст Юникода).

Кроме того, при работе с текстовыми файлами выполняется обязательное декодирование их содержимого в Юникод в соответствии с выбранной кодировкой, поэтому вам придется использовать двоичный режим

для чтения содержимого файлов, не поддающихся декодированию, в виде строк байтов (или обрабатывать исключения декодирования в Юникод с помощью инструкций `try` и пропускать такой файл целиком). Это относится и к собственно двоичным файлам, и к текстовым файлам, для представления текста в которых используется неподдерживаемая или неизвестная кодировка. Как мы увидим далее в этой главе, в версии 3.X строки типа `str` всегда содержат текст Юникода, поэтому иногда придется использовать строки байтов для представления имен файлов при использовании таких инструментов, как `os.listdir`, `glob.glob` и `os.walk`, если они не могут быть декодированы (передача в виде строки байтов фактически подавляет необходимость декодирования).

На протяжении всей книги мы будем видеть примеры влияния различий между текстовым и двоичным типами `str` и `bytes` в инструментах для работы с файлами: в главах 5 и 12, когда будем исследовать сокеты; в главах 6 и 11, когда нам потребуется игнорировать ошибки Юникода при поиске в файлах и каталогах; в главе 12, когда будем знакомиться с модулями поддержки протоколов Интернета на стороне клиента, таких как FTP и протоколы электронной почты, реализованные поверх сокетов, предполагающих определение режимов файлов и кодировок; и и во многих других местах.

Но так же, как и для строковых типов, в данной главе мы не будем углубляться в эту тему, хотя и будем рассматривать практическое влияние некоторых из представленных концепций. Файлы и строки являются базовой частью языка, и знание их является необходимым условием для чтения этой книги. Как упоминалось ранее, поддержке Юникода посвящена 45-страничная глава в четвертом издании книги «Изучаем Python», поэтому я не буду повторять эти сведения в данной книге. Если при чтении следующих разделов вам покажется, что вы вконец запутались в концепциях, связанных с Юникодом, и в различиях между текстовыми и двоичными строками и файлами, я советую обратиться за более полной информацией к указанной выше книге или к другим источникам.

Использование встроенных объектов файлов

Несмотря на различия между текстовыми и двоичными данными в Python 3.X, файлы по-прежнему очень просты в использовании. Для большинства задач обработки файлов в сценариях достаточно знать функцию `open`. Объект файла, возвращаемый функцией `open`, обладает методами для чтения данных (`read`, `readline`, `readlines`), записи данных (`write`, `writelines`), освобождения системных ресурсов (`close`), перемещения по файлу (`seek`), принудительного выталкивания выходных буферов на диск (`flush`), получения соответствующего дескриптора файла (`fileno`) и других. Но так как встроенный объект файла очень прост в использовании, давайте сразу рассмотрим несколько интерактивных примеров.

Вывод в файлы

Чтобы создать новый файл, следует вызвать функцию `open` с двумя аргументами: внешним *именем* создаваемого файла и строкой *режима* `"w"` (от *write* – запись). Чтобы сохранить данные в файле, нужно вызвать метод `write` объекта файла со строкой, содержащей данные, которые нужно сохранить, а затем метод `close`, чтобы закрыть файл. Метод `write` вернет количество символов или байтов, записанных в файл (о котором мы не всегда будем упоминать для экономии места в книге). Вызов метода `close`, как мы увидим далее, не является обязательным, если вам требуется открыть и прочитать файл повторно в той же программе или сеансе:

```
C:\temp> python
>>> file = open('data.txt', 'w')       # откроет файл для вывода: создаст объект
>>> file.write('Hello file world!\n')  # запишет строку, как есть
18
>>> file.write('Bye   file world.\n')  # вернет число символов/байтов
18
>>> file.close()                       # закрытие "сборщиком мусора" и выход
```

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 этот файл также будет виден в менеджере файлов с графическим интерфейсом.

Открытие файлов. В вызове функции `open`, показанном в предыдущем примере, первый аргумент может содержать необязательный полный путь к файлу. Если просто передать имя файла без указания пути, файл окажется в текущем рабочем каталоге Python. То есть он появится в том месте, откуда был запущен программный код, – в данном случае простое имя файла *data.txt* предполагает использование каталога *C:\temp* на моем компьютере, поэтому в реальности будет создан файл *C:\temp\data.txt*. Если быть более точным, в случае отсутствия абсолютного пути в имени файла путь к нему определяется относительно текущего рабочего каталога. Освежить эту тему в памяти можно с помощью раздела «Текущий рабочий каталог» (глава 3).

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 присутствует (поэтому следует проявлять осторожность – при открытии в этом режиме вы потеряете все, что находилось в файле).

Запись. Обратите внимание, что в строки, записываемые в файл, был явно добавлен символ конца строки `\n`. В отличие от функции `print`, метод `write` объекта файла записывает в точности то, что ему передано, без дополнительного форматирования. Строка, переданная методу `write`, появляется во внешнем файле «символ в символ». При записи в текстовые файлы может выполняться преобразование символов конца строки или операция кодирования Юникода, о которых упоминалось выше, а когда позднее данные будут читаться из файла, автоматически будут выполнены обратные преобразования.

Для записи в файлы можно также использовать метод `writelines`, который просто записывает все строки из списка без дополнительного форматирования. Например, ниже приводится вызов `writelines`, эквивалентный двум вызовам `write`, показанным ранее:

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

Этот метод используется не очень часто (и может эмулироваться простым циклом `for` и другими инструментами итераций), но его удобно использовать в сценариях, которые сначала сохраняют выходные данные в списке, а потом записывают его в файл.

Закрытие. Использованный выше метод файла `close` завершает формирование содержимого файла и освобождает системные ресурсы. Например, закрытие файла влечет выталкивание на диск буферизованных выходных данных. Обычно файлы автоматически закрываются, когда объект файла уничтожается интерпретатором при сборке мусора (то есть когда в сценарии исчезнет последняя ссылка на объект). Под этим подразумеваются все файлы, оставшиеся открытыми к моменту завершения программы или сеанса Python. По этой причине вызов метода `close` часто является необязательным. На практике часто можно увидеть программный код, обрабатывающий файлы, который использует эту идиому:

```
open('somefile.txt', 'w').write("G'day Bruce\n") # записать во временный файл
open('somefile.txt', 'r').read()                 # прочитать временный файл
```

Так как в обоих выражениях создаются временные объекты файлов, сразу же осуществляется запись или чтение и при этом не сохраняются ссылки на них, объекты файлов немедленно автоматически закрываются и уничтожаются сразу после выполнения операции. В таких случаях нет никакой необходимости вызывать метод `close` явно.

Однако в некоторых контекстах вам может потребоваться явно закрывать файлы:

- Во-первых, реализация Jython опирается на механизм сборки мусора в интерпретаторе Java, поэтому вы не всегда можете знать, когда файлы будут закрыты, как вы это знаете при работе со стандартным Python. Если вы собираетесь запускать свой программный код на языке Python под управлением Jython, вам может потребоваться за-

крывать файлы вручную, если программа создает большое количество объектов файлов за короткое время (например, в цикле), чтобы избежать исчерпания файловых ресурсов в операционных системах, где их количество ограничено.

- Во-вторых, некоторые среды разработки, такие как стандартный графический интерфейс IDLE, могут удерживать объекты файлов дольше, чем хотелось бы (например, в объектах с трассировочной информацией о предыдущих ошибках), и тем самым препятствовать немедленной их утилизации сборщиком мусора. Выполняя запись в выходной файл в среде IDLE, обязательно закрывайте его явно (или вызывайте метод `flush`), если вам необходимо обеспечить достоверное чтение информации из этого файла в течение того же сеанса IDLE. В противном случае может получиться, что выходные буферы не будут вытеснены на диск, и при чтении вы получите неполные данные.
- И хотя это и кажется маловероятным, тем не менее такая особенность, как автоматическое закрытие файлов, в будущем может измениться. Технически это особенность реализации объектов файлов, которая с течением времени может перестать рассматриваться как часть определения языка.

По этим причинам закрытие файлов вручную – хороший навык для нетривиальных программ, даже если с технической точки зрения это не требуется. Закрытие файлов вообще не только не нанесет вреда, но и считается полезной практикой, которой стоит следовать.

Гарантированное закрытие файлов: обработчики исключений и менеджеры контекста

Операция закрытия файла вручную оформляется в программном коде очень просто, но как гарантировать закрытие файла в случае исключения, которое может отбросить поток выполнения программы далеко от того места, где находится вызов метода `close`? Прежде всего, не забывайте, что файлы закрываются сами, когда соответствующие им объекты файлов утилизируются сборщиком мусора, причем это произойдет в любом случае, даже если возникнет исключение.

Однако если необходимо обеспечить явное закрытие файла в любом случае, у вас есть два пути: наиболее типичный – использование инструкции `try` с предложением `finally`, потому что оно позволяет реализовать выполнение заключительных операций для любых типов исключений:

```
myfile = open(filename, 'w')
try:
    ...обработка myfile...
finally:
    myfile.close()
```

В последних версиях Python появилась инструкция with, обеспечивающая более краткий способ реализации заключительных операций для объектов определенных типов, включая закрытие файлов:

```
with open(filename, 'w') as myfile:
    ... обработка myfile, закрывается автоматически после выхода...
```

Данная инструкция опирается на использование менеджера контекста объекта файла: программного кода, который автоматически вызывается при входе и при выходе из инструкции, независимо от того, возникло ли исключение. Поскольку реализация выхода в объекте файла автоматически закрывает файл, этот прием гарантирует закрытие файла в любом случае, независимо от возникновения исключения.

Решение на основе инструкции with выглядит заметно короче (на 3 строки), чем альтернативное решение на основе конструкции try/finally, но оно является менее универсальным – инструкция with может применяться только к объектам, поддерживающим протокол менеджеров контекста, тогда как конструкция try/finally позволяет реализовать произвольные заключительные операции для произвольных контекстов исключений. Область применения инструкции with ограничена, несмотря на то, что у некоторых типов объектов также имеются менеджеры контекста (например, у блокировок потоков). Если вам хочется помнить только один вариант реализации заключительных операций, то конструкция try/finally выглядит наиболее объемлющей. При этом инструкция with позволяет уменьшить объем программного кода для файлов, которые должны быть закрыты в любом случае, и прекрасно справляется с этой конкретной задачей. Она позволяет сэкономить строку программного кода, когда обработка исключений не предусматривается (хотя и за счет добавления в логику обработки файла еще одного уровня вложенности и отступов):

```
myfile = open(filename, 'w')   # традиционная форма
...обработка myfile...
myfile.close()

with open(filename) as myfile: # с применением менеджера контекста
    ... обработка myfile...
```

В версии Python 3.1 и выше эта инструкция позволяет указывать несколько (то есть вложенные) менеджеров контекста – в инструкции можно перечислить через запятую любое количество менеджеров контекста, и она будет действовать, как набор вложенных друг в друга инструкций with. В общем случае в версии 3.1 и выше следующий программный код:

```
with A() as a, B() as b:
    ...инструкции...
```

действует так же, как программный код ниже, который можно использовать в версиях 3.1, 3.0 и 2.6:

```
with A() as a:
    with B() as b:
        ...инструкции...
```

Например, когда программа выходит из следующего блока инструкции with, автоматически выполняются действия по закрытию обоих файлов, независимо от того, возникло исключение или нет:

```
with open('data') as fin, open('results', 'w') as fout:
    for line in fin:
        fout.write(transform(line))
```

В последние годы такой программный код, опирающийся на использование менеджеров контекста, становится все более привычным, причем отчасти благодаря приходу новых программистов из языков, требующих вручную закрывать файлы в любых случаях. В большинстве ситуаций нет никакой необходимости обертывать инструкциями with программный код обработки файлов – часто бывает вполне достаточно того, что объекты файлов автоматически закрываются при утилизации, а для других ситуаций достаточно вручную вызывать метод close. Приемы, основанные на использовании инструкций with и try, описанные выше, следует использовать только в случае необходимости явно закрывать файлы и только, когда существует вероятность исключений. Поскольку стандартная реализация C Python автоматически закрывает файлы при утилизации объектов, во многих (если не в большинстве) ситуациях ни один из приведенных вариантов не является необходимым.

Чтение из файлов

Чтение данных из внешних файлов осуществляется столь же просто, как запись, но при этом доступно большее количество методов, позволяющих загружать данные в разнообразных режимах. Входные текстовые файлы открываются с флагом режима "r" (от «read» – читать) либо вообще без флага режима ("r" – значение по умолчанию, и параметр часто пропускается). После открытия текстового файла его строки можно читать с помощью метода readlines:

```
C:\temp> python
>>> file = open('data.txt')    # открыть входной файл: 'r' - по умолчанию
>>> lines = file.readlines()   # прочитать в список строк
>>> for line in lines:         # НО! использовать итератор файла!
...     print(line, end='')    # строки оканчиваются символом '\n'
...
Hello file world!
Bye   file world.
```

Метод readlines загружает содержимое файла в память целиком и передает 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 символы (или байты), хранящиеся в файле.

`file.read(N)`

Возвращает строку, содержащую очередные N символов (или байтов) из файла.

`file.readline()`

Читает содержимое файла до ближайшего символа `\n` и возвращает строку.

`file.readlines()`

Читает файл целиком и возвращает список строк.

Попробуем воспользоваться этими методами для чтения файлов, строк и символов из файлов – вызов метода `seek(0)` перед каждой попыткой чтения переустанавливает текущую позицию чтения в начало файла (подробнее об этом методе рассказывается чуть ниже):

```
>>> file.seek(0)              # перейти в начало файла
>>> file.read()               # прочитать в строку файл целиком
'Hello file world!\nBye file world.\n'

>>> file.seek(0)
>>> file.readlines()          # прочитать файл целиком в список строк
['Hello file world!\n', 'Bye   file world.\n']

>>> file.seek(0)
>>> file.readline()           # читать по одной строке
'Hello file world!\n'
>>> file.readline()
'Bye file world.\n'
>>> file.readline()           # конец файла - возвращается пустая строка
''

>>> file.seek(0)              # прочитать N (или оставшиеся) символы/байты
>>> file.read(1), file.read(8) # конец файла - возвращается пустая строка
('H', 'ello fil')
```

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

- `read()` и `readlines()` загружают в память сразу *весь файл*. Это удобно, когда желательно получить содержимое файла, написав более короткий программный код. Кроме того, эти методы действуют очень быстро, но для больших файлов их применение накладно: загрузка гигабайтных файлов – обычно не самое лучшее решение (а кроме того, на некоторых компьютерах – просто невозможное).

- С другой стороны, вызовы `readline()` и `read(N)` возвращают лишь *часть файла* (очередную строку или блок из N символов или байтов), поэтому они надежнее для потенциально больших файлов, но не так удобны и обычно работают медленнее. Оба метода возвращают пустую строку по достижении конца файла. Если скорость для вас важна, а ваши файлы не слишком велики, методы `read` и `readlines` могут оказаться лучшим выбором.
- Кроме того, смотрите обсуждение итераторов файлов в следующем разделе. Как мы увидим, итераторы объединяют в себе удобство метода `readlines()` и экономное отношение к памяти метода `readline()`, и на сегодняшний день являются наиболее предпочтительным способом построчного чтения текстовых файлов.

Часто встречающийся здесь вызов `seek(0)` означает «вернуться в начало файла». В нашем примере этот вызов является альтернативой повторному открытию файла перед очередной попыткой. Все операции чтения и записи в файлах происходят в текущей позиции. Обычно при открытии текущая позиция в файле устанавливается со смещением 0 и перемещается вперед по мере передачи данных. Метод `seek` просто позволяет переместиться в новую позицию для очередной операции передачи данных. Подробнее об этом методе будет рассказываться ниже, когда мы перейдем к исследованию возможности произвольного доступа к файлам.

Чтение строк с помощью итераторов файлов

В прежних версиях Python принятым способом построчного чтения информации из файла в цикле `for` было чтение файла в список, а затем обход этого списка в цикле:

```
>>> file = open('data.txt')
>>> for line in file.readlines():  # НЕ ДЕЛАЙТЕ ТАК БОЛЬШЕ!
...     print(line, end='')
```

Если вы уже изучили основы языка с помощью других книг, таких как «Изучаем Python», возможно, вы знаете, что того же результата можно добиться с меньшими усилиями – и для вас, и для вашего компьютера. В последних версиях Python объект файла включает итератор, который при каждом обращении извлекает только одну строку из файла в любых итерационных контекстах, включая циклы `for` и генераторы списков. Практическая выгода заключается в том, что теперь нет необходимости вызывать метод `readlines` в цикле `for`, чтобы построчно просканировать содержимое файла, – итератор читает строки автоматически:

```
>>> file = open('data.txt')
>>> for line in file:          # нет необходимости вызывать readlines
...     print(line, end='')    # итератор каждый раз читает следующую строку
...
Hello file world!
Bye   file world.
```

Более того – теперь файл можно открывать непосредственно в инструкции цикла, как временный, который будет автоматически закрыт сборщиком мусора после выхода из цикла (так как часто цикл – это единственная ссылка на объект файла):

```
>>> for line in open('data.txt'):  # еще короче: временный объект файла
...     print(line, end='')        # будет закрыт при утилизации автоматически
...
Hello file world!
Bye   file world.
```

Кроме того, такая форма обхода строк в файле не вызывает загрузку всего содержимого файла в список строк, поэтому она более экономно расходует память при работе с большими текстовыми файлами. По этой причине данный способ построчного чтения файлов является наиболее предпочтительным на сегодняшний день. Если вам интересно узнать, что же в действительности происходит внутри цикла `for`, вы можете попробовать использовать итератор вручную. Итератор – это всего лишь метод `__next__` (вызываемый встроенной функцией `next`), который своим поведением напоминает метод `readline`, за исключением того, что по достижении конца файла методы чтения возвращают пустую строку, а итератор возбуждает исключение, чтобы прервать итерации:

```
>>> file = open('data.txt')    # методы чтения: пустая строка в конце файла
>>> file.readline()
'Hello file world!\n'
>>> file.readline()
'Bye   file world.\n'
>>> file.readline()
''


>>> file = open('data.txt')    # итераторы: исключение в конце файла
>>> file.__next__()            # не нужно предварительно вызывать iter(file),
'Hello file world!\n'          # потому что файлы имеют собственные итераторы
>>> file.__next__()
'Bye   file world.\n'
>>> file.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Интересно отметить, что итераторы автоматически используются во всех итерационных контекстах, включая конструктор списка, генераторы списков, функцию `map` и оператор `in` проверки на вхождение:

```
>>> open('data.txt').readlines()                          # всегда читает строки
['Hello file world!\n', 'Bye   file world.\n']

>>> list(open('data.txt'))                                # выполняет обход строк
['Hello file world!\n', 'Bye   file world.\n']

>>> lines = [line.rstrip() for line in open('data.txt')]  # генераторы
```

```
>>> lines
['Hello file world!', 'Bye   file world.']

>>> lines = [line.upper() for line in open('data.txt')]  # произв. действия
>>> lines
['HELLO FILE WORLD!\n', 'BYE   FILE WORLD.\n']

>>> list(map(str.split, open('data.txt')))               # применение функции
[['Hello', 'file', 'world!'], ['Bye', 'file', 'world.']]

>>> line = 'Hello file world!\n'
>>> line in open('data.txt')                             # проверка на вхождение
True
```

На первый взгляд итераторы могут показаться не слишком представительными, но они предоставляют множество способов, упрощающих жизнь разработчиков программ на языке Python.

Другие режимы открытия файлов

Помимо режимов открытия файлов "w" и "r" (по умолчанию) большинством платформ поддерживается строка режима открытия "a", означающая «append» (дополнение). В этом режиме вывода методы записи добавляют данные в конец файла, и вызов функции open не уничтожает текущее содержимое файла:

```
>>> file = open('data.txt', 'a')     # для дополнения: содержимое не стирается
>>> file.write('The Life of Brian')  # добавит в конец существующих данных
>>> file.close()
>>>
>>> open('data.txt').read()          # открыть и прочитать весь файл
'Hello file world!\nBye   file world.\nThe Life of Brian'
```

Хотя в большинстве случаев для открытия файлов применяются уже рассмотренные нами формы вызовов, но кроме того, функция open может принимать дополнительные аргументы, позволяющие более точно определить потребности обработки файла. Чаще всего в практике используются первые три аргумента – имя файла, режим открытия и размер буфера. Все они, кроме первого, являются необязательными: если они опущены, принимается режим открытия по умолчанию "r" (ввод) и разрешается полная буферизация. Ниже приводятся некоторые сведения об этих трех аргументах функции open, которые вам следует знать:

Имя файла

Как уже говорилось, имена файлов могут включать путь к каталогу, что дает возможность ссылаться на файлы, находящиеся на компьютере в произвольном месте; если полный путь в имени файла отсутствует, считается, что путь к файлам указывается относительно текущего рабочего каталога (который описывался в предыдущей главе). В целом, любой формат имени файла, который можно ввести

в системной оболочке, можно использовать и в вызове функции `open`. Например, аргумент имени файла `r'..\temp\spam.txt'` в Windows соответствует файлу *spam.txt* в подкаталоге *temp*, находящемся в родительском каталоге текущего рабочего каталога, – на один шаг вверх и затем вниз в каталог *temp*.

Режим открытия

Функция `open` может принимать и другие режимы, часть из которых мы увидим далее в этой главе, `"r+"`, `"w+"` и `"a+"`, которые используются, чтобы открыть файл для чтения *и* записи, и `"b"` – для обозначения двоичного режима. В частности, режим `"r+"` означает, что файл доступен как для чтения, так и для записи, при этом содержимое существующих файлов сохраняется; `"w+"`, позволяет выполнять операции чтения и записи, но создает файл заново, уничтожая прежнее его содержимое; режимы `"rb"` и `"wb"` разрешают читать и записывать данные в двоичном режиме без выполнения автоматических преобразований; наконец, режимы `"wb+"` и `"r+b"` объединяют возможность чтения и записи с двоичным режимом. Проще говоря, по умолчанию используется режим для чтения `"r"`, но вы можете использовать режим `"w"` для записи и `"a"` для дополнения, можете добавлять символ `+`, чтобы обеспечить возможность изменения содержимого файла, а также указывать `b` и `t`, чтобы задать двоичный или текстовый режим. Порядок следования спецификаторов в строке режима не имеет значения.

Как будет показано ниже в этой главе, режимы со спецификатором `+` часто используются совместно с методом `seek`, обеспечивающим возможность произвольного доступа к файлам. Независимо от режима содержимым файлов в программах Python всегда являются строки – методы чтения возвращают строку, и строку мы передаем методам записи. Однако тип используемой строки зависит от выбранного режима: `str` – для текстового режима, и `bytes` или другие типы строк байтов – для двоичного режима.

Размер буфера

Функция `open` также принимает необязательный третий аргумент с размером буфера, позволяющий управлять буферизацией файла – способом размещения данных в очереди, позволяющим повысить производительность. Значение 0 в этом аргументе означает отсутствие буферизации (данные передаются немедленно, но это значение допустимо только для двоичных режимов), значение 1 означает построчную буферизацию, а любое другое положительное число означает использование режима полной буферизации (который используется по умолчанию, если третий аргумент отсутствует в вызове функции).

Как обычно, полные сведения о дополнительных аргументах функции `open`, помимо этих трех, вы найдете в руководстве по библиотеке языка Python и в справочных изданиях. Например, функция `open` может при-

нимать дополнительные аргументы, управляющие поведением отображения *конца файла* и определяющие *кодировку символов*, автоматически применяемую к содержимому текстовых файлов. Обе эти концепции будут обсуждаться в следующем разделе, поэтому двинемся дальше.

Двоичные и текстовые файлы

Во всех предыдущих примерах обрабатываются простые текстовые файлы, но сценарии на языке Python могут также открывать и обрабатывать файлы, содержащие *двоичные* данные – изображения JPEG, аудиоклипы, упакованные двоичные данные, произведенные программами на языке FORTRAN и C, кодированный текст и все остальное, что может храниться в файлах в виде последовательностей байтов. Главное отличие для программного кода заключается в аргументе *режима*, передаваемом встроенной функции `open`:

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

После открытия двоичных файлов таким способом можно читать и записывать их содержимое с помощью представленных выше методов: `read`, `write` и так далее. Методы `readline` и `readlines`, как и построчные итераторы файлов, по-прежнему будут работать с текстовыми файлами, открытыми в двоичном режиме, но нет никакого смысла применять их к действительно двоичным данным, которые не имеют построчной организации (байты, обозначающие конец строки в текстовых данных, не имеют такого смысла в двоичных данных, да и вообще их может не быть в файле).

Во всех случаях данные, перемещаемые между файлами и программами, представляются в сценариях в виде строк Python, даже если они являются двоичными. Однако для файлов, открытых в двоичном режиме, содержимое файла будет представлено в виде *строк байтов*. Продолжим предыдущий пример:

```
>>> open('data.txt').read()        # текстовый режим: тип str
'Hello file world!\nBye   file world.\nThe Life of Brian'

>>> open('data.txt', 'rb').read()  # двоичный режим: тип bytes
b'Hello file world!\r\nBye   file world.\r\nThe Life of Brian'

>>> file = open('data.txt', 'rb')
>>> for line in file: print(line)
...
b'Hello file world!\r\n'
b'Bye   file world.\r\n'
b'The Life of Brian'
```

Это обусловлено тем, что в Python 3.X содержимое текстовых файлов интерпретируется, как последовательность символов Юникода, которая автоматически декодируется при чтении и кодируется при записи.

Содержимое файлов, открытых в двоичном режиме, напротив, доступно в виде простых строк байтов, для которых никаких промежуточных преобразований не выполняется, – они содержат именно то, что хранится в файле. В Python 3.X строки типа `str` всегда содержат символы Юникода, поэтому для представления двоичных данных потребовалось ввести специальный строковый тип `bytes`, представляющий последовательность однобайтовых целых чисел, которые могут иметь любые 8-битовые значения. Обычные строки и строки байтов обладают практически идентичными наборами операций, поэтому различия между ними в большинстве случаев незаметны, но имейте в виду, что действительно двоичные файлы для чтения *должны* открываться в двоичном режиме, потому что они могут содержать данные, которые невозможно будет декодировать в текст Юникода.

Точно так же при выводе в двоичные файлы необходимо использовать строки байтов, потому что обычные строки интерпретируются не как двоичные данные, а как декодированные символы Юникода (то есть кодовые пункты), которые должны быть закодированы в двоичное представление при записи в файл в двоичном или текстовом режиме:

```
>>> open('data.bin', 'wb').write(b'Spam\n')
5

>>> open('data.bin', 'rb').read()
b'Spam\n'

>>> open('data.bin', 'wb').write('spam\n')
TypeError: must be bytes or buffer, not str
(TypeError: аргумент должен иметь тип bytes или buffer, но не str)
```

Но обратите внимание, что в данном примере строки завершаются символом `\n` вместо последовательности `\r\n`, принятой в Windows, которая присутствовала в предыдущем примере работы с текстовым файлом в двоичном режиме. Строго говоря, двоичный режим запрещает не только кодирование символов Юникода, но и автоматическое преобразование символов конца строки, которые по умолчанию выполняются для файлов, открытых в текстовом режиме. Но прежде чем нам удастся полностью с этим разобраться, необходимо познакомиться с двумя основными отличиями между текстовыми и двоичными файлами.

Кодирование символов Юникода в текстовых файлах

Как упоминалось выше, объекты файлов, открытые в текстовом режиме, при передаче данных между программой и внешним файлом всегда преобразуют данные в соответствии с кодировкой по умолчанию или указанной явно. При записи в файл выполняется кодирование данных, а при чтении – декодирование. Для файлов, открытых в двоичном режиме, никаких преобразований не выполняется, потому что именно это и требуется для действительно двоичных данных. Например, взгляните на следующую строку, содержащую символ Юникода, двоичное

представление которого выходит за рамки 7-битового диапазона представления символов ASCII:

```
>>> data = 'sp\xe4m'
>>> data
'späm'
>>> 0xe4, bin(0xe4), chr(0xe4)
(228, '0b11100100', 'ä')
```

Эту строку можно закодировать вручную в соответствии с той или иной кодировкой, и для разных кодировок будут получаться различные двоичные представления строки:

```
>>> data.encode('latin1')      # 8-битовые символы: ascii + дополнительные
b'sp\xe4m'

>>> data.encode('utf8')        # 2 байта отводится только
b'sp\xc3\xa4m'                 # для специальных символов

>>> data.encode('ascii')       # кодирование в ascii невозможно
UnicodeEncodeError: 'ascii' codec can't encode character '\xe4' in position 2:
ordinal not in range(128)
```

(UnicodeEncodeError: кодек 'ascii' не может преобразовать символ '\xe4' в позиции 2: число выходит за пределы range(128))

Интерпретатор Python отображает печатаемые символы в таких строках как обычно, а непечатаемые – в виде шестнадцатеричных экранированных значений `\xNN`, количество которых увеличивается при использовании некоторых более сложных схем кодирования (`cp500` в следующем примере – это кодировка EBCDIC):

```
>>> data.encode('utf16')       # по 2 байта на символ плюс преамбула
b'\xff\xfes\x00p\x00\xe4\x00m\x00'

>>> data.encode('cp500')       # кодировка ebcdic: двоичное представление
b'\xa2\x97C\x94'               # строки существенно отличается
```

Результат кодирования здесь отражает двоичное представление строки, которое будет записано в файл при сохранении. Однако выполнять кодирование вручную обычно не требуется, потому что для текстовых файлов кодирование выполняется автоматически при передаче данных – данные декодируются при чтении и кодируются при записи, в соответствии с именем указанной кодировки (или с использованием кодировки, используемой на текущей платформе: смотрите описание функции `sys.getdefaultencoding`). Продолжим интерактивный сеанс:

```
>>> open('data.txt', 'w', encoding='latin1').write(data)
4
>>> open('data.txt', 'r', encoding='latin1').read()
'späm'
>>> open('data.txt', 'rb').read()
b'sp\xe4m'
```

Если файл открыть в двоичном режиме, никаких преобразований производиться не будет – последняя инструкция в предыдущем примере предъявляет в точности то, что хранится в файле. Чтобы увидеть отличия при использовании других кодировок, сохраним эту строку еще раз:

```
>>> open('data.txt', 'w', encoding='utf8').write(data) # кодировка utf8
4
>>> open('data.txt', 'r', encoding='utf8').read() # декодирование: отменяет
'späm'                                          # кодирование
>>> open('data.txt', 'rb').read()               # преобразование
b'sp\xc3\xa4m'                                  # не производится
```

На этот раз двоичное содержимое файла получилось другим, но в результате автоматического декодирования, которое выполняется при чтении файла в текстовом режиме, возвращается та же самая строка. В действительности, кодировка имеет значение для строк, только когда они находятся в файлах, – сразу после загрузки в память строки превращаются в простые последовательности символов Юникода («кодовые пункты»). Этот этап преобразования желателен для текстовых файлов, но не для двоичных. При использовании двоичных режимов этап преобразования содержимого пропускается, поэтому при работе с истинно двоичными данными необходимо использовать эти режимы. Если вам нужны доказательства, попробуйте сами: попытка записать или прочитать недекодируемые данные в текстовом режиме приведет к появлению ошибки:

```
>>> open('data.txt', 'w', encoding='ascii').write(data)
UnicodeEncodeError: 'ascii' codec can't encode character '\xe4' in position 2:
ordinal not in range(128)
```

(UnicodeEncodeError: кодек 'ascii' не может преобразовать символ '\xe4' в позиции 2: число выходит за пределы range(128))

```
>>> open(r'C:\Python31\python.exe', 'r').read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2:
character maps to <undefined>
```

(UnicodeDecodeError: кодек 'charmap' не может преобразовать байт 0x90 в позиции 2: символ отображается в символ <undefined>)

Двоичный режим можно также рассматривать, как последний шанс прочитать текстовый файл, если он не может быть декодирован с использованием кодировки по умолчанию, а кодировка файла неизвестна. Следующий программный код воссоздает оригинальные строки, когда кодировка известна, но терпит неудачу, когда она неизвестна, если только не использовать двоичный режим (такие ошибки могут возникать как при чтении данных, так и при записи, но в любом случае программный код терпит неудачу):

```
>>> open('data.txt', 'w', encoding='cp500').writelines(['spam\n', 'ham\n'])
>>> open('data.txt', 'r', encoding='cp500').readlines()
```

```
['spam\n', 'ham\n']

>>> open('data.txt', 'r').readlines()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x81 in position 2:
character maps to <undefined>
(UnicodeDecodeError: кодек 'charmap' не может преобразовать байт 0x81 в позиции
2: символ отображается в символ <undefined> )

>>> open('data.txt', 'rb').readlines()
[b'\xa2\x97\x81\x94\r%\x88\x81\x94\r%']

>>> open('data.txt', 'rb').read()
b'\xa2\x97\x81\x94\r%\x88\x81\x94\r%'
```

Если вы имеете дело только с текстом ASCII, вы можете пропустить все, что связано с кодировками, – данные в файлах будут один-в-один отображаться в символы в строках, потому что ASCII является подмножеством большинства кодировок, используемых по умолчанию. Если вам приходится обрабатывать файлы, созданные с применением других кодировок, и, возможно, на других платформах (например, файлы, полученные из Интернета), вам может потребоваться использовать двоичный режим, если кодировка заранее не известна. Однако имейте в виду, что текст в кодированном двоичном представлении не может обрабатываться так, как вам хотелось бы: текст, закодированный с применением определенной кодировки, не может сравниваться или объединяться с текстом, закодированным с применением других кодировок.

И снова за дополнительной информацией о Юникоде обращайтесь к другим ресурсам. Мы еще не раз будем возвращаться к теме Юникода в этой книге: в главе 9 будет показано, какое влияние оказывает Юникод на виджет `Text` из библиотеки tkinter, а в части IV, охватывающей вопросы программирования для Интернета, мы узнаем, как это отражается на данных, доставляемых по сети с использованием протоколов FTP, электронной почты и в Интернете в целом. Текстовые файлы обладают еще одной особенностью, отсутствующей у двоичных файлов: преобразование символов конца строки, что является темой следующего раздела.

Преобразование символов конца строки в Windows

По историческим причинам конец строки текста в файле представляется на разных платформах различными символами. В Unix и Linux – это одиночный символ `\n`, а в Windows – это последовательность из двух символов `\r\n`. В результате файлы, перемещаемые между Linux и Windows, могут после передачи странно выглядеть в текстовом редакторе – они могут сохранить окончание строки, принятое на исходной платформе.

Например, большинство текстовых редакторов для Windows обрабатывает текст в формате Unix, но Блокнот (Notepad) составляет заметное ис-

ключение – текстовые файлы, скопированные из Unix или Linux, обычно выглядят в Блокноте, как одна большая строка со странными символами внутри (\n). Точно так же при копировании файлов из Windows в Unix в двоичном режиме в них сохраняется символ \r (который в текстовых редакторах часто отображается как ^M).

Сценариям на языке Python это обычно безразлично, потому что объекты файлов автоматически отображают последовательность DOS \r\n в одиночный символ \n. При выполнении сценариев в Windows это действует так:

- Для файлов, открытых в текстовом режиме, при чтении \r\n преобразуется в \n.
- Для файлов, открытых в текстовом режиме, при записи \n преобразуется в \r\n.
- Для файлов, открытых в двоичном режиме, никакие преобразования не производятся.

В Unix-подобных системах преобразование не производится в любом режиме, потому что в файлах используется символ \n. Следует запомнить два важных следствия из этих правил. Во-первых, почти всегда во всех сценариях на языке Python символ конца строки представляется одиночным \n, независимо от способа его сохранения во внешних файлах на соответствующей платформе. Путем соответствующего преобразования \n при чтении и записи Python скрывает присущие платформам различия.

Второе следствие из этого преобразования более тонкое: при обработке двоичных файлов использование двоичного режима (например, rb, wb) отключает механизм преобразования символов конца строки. Если выбрать неправильный режим, указанные преобразования вполне могут повредить данные, как при чтении, так и при записи, – случайно оказавшиеся среди двоичных данных байты \r могут быть ошибочно отброшены при чтении или ошибочно добавлены к байтам \n при записи. В итоге двоичные данные окажутся искаженными, что, вероятно, совсем не то, что вам хотелось бы получить при работе с изображениями или аудиоклипами!

В Python 3.X эта проблема ушла на задний план, потому что мы в принципе не можем использовать двоичные данные с файлами, открытыми в текстовом режиме, из-за того, что текстовый режим предполагает автоматическое применение кодировок Юникода к содержимому файлов. Операции чтения и записи просто будут терпеть неудачу, если данные не смогут быть декодированы при чтении или закодированы при записи. Использование двоичного режима позволяет избежать ошибок, связанных с преобразованием Юникода, и автоматически запрещает преобразование символов конца строки как таковое (ошибки, связанные с преобразованием Юникода, можно было бы перехватывать в инструкции try). Итак, стоит запомнить как отдельный факт, что двоичный ре-

жим предохраняет двоичные данные от искажения в результате преобразования символов конца строки, особенно если вы работаете только с текстовыми данными ASCII, когда можно смело забыть обо всех проблемах, связанных с Юникодом.

Ниже демонстрируется действие механизма преобразования символов конца строки в Python 3.1 в Windows – объект файла, открытого в текстовом режиме, выполняет преобразование символов конца строки и обеспечивает переносимость наших сценариев:

```
>>> open('temp.txt', 'w').write('shrubbery\n') # запись в текстовом режиме:
10                                             # \n -> \r\n
>>> open('temp.txt', 'rb').read()              # чтение двоичных данных:
b'shrubbery\r\n'                               # фактические байты из файла
>>> open('temp.txt', 'r').read()               # проверка чтением: \r\n -> \n
'shrubbery\n'
```

При записи в двоичном режиме, напротив, предотвращаются любые преобразования, как и предполагалось, даже когда данные содержат байты, которые в текстовом режиме интерпретировались бы как часть символов конца строки (при выводе строк байтов отдельные байты выводятся как символы ASCII, если они соответствуют печатаемым символам, и как экранированные шестнадцатеричные последовательности в противном случае):

```
>>> data = b'a\0b\rc\r\nd'             # 4 байта, 4 обычных символа
>>> len(data)
8
>>> open('temp.bin', 'wb').write(data) # запись двоичных данных как есть
8
>>> open('temp.bin', 'rb').read()      # чтение двоичных данных:
b'a\x00b\rc\r\nd'                      # без преобразования
```

Но при чтении двоичных данных в текстовом режиме, неважно, случайно или нет, механизм преобразования символов конца строки может повредить данные (предполагая, что вообще при декодировании не возникло ошибок, как с нашими ASCII-данными на платформе Windows):

```
>>> open('temp.bin', 'r').read() # чтение в текстовом режиме: искажены \r!
'a\x00b\nc\nd'
```

Тот же эффект может возникнуть при записи двоичных данных в текстовом режиме – могут изменяться или вставляться байты, значения которых совпадают с символами конца строки (и снова имеются в виду данные, успешно прошедшие этап кодирования в кодировку по умолчанию для текущей платформы):

```
>>> open('temp.bin', 'w').write(data)  # в текстовом режиме должна
TypeError: must be str, not bytes      # передаваться строка типа str
                                       # используйте bytes.decode()
                                       # для преобразования типа
```

```
>>> data.decode()
'a\x00b\rc\r\nd'
>>> open('temp.bin', 'w').write(data.decode())
8
>>> open('temp.bin', 'rb').read()      # запись в текстовом режиме: добавит \r
b'a\x00b\rc\r\r\nd'

>>> open('temp.bin', 'r').read()       # опять искажение, изменит \r
'a\x00b\nc\n\nd'
```

Проще говоря, запомните, что во всех текстовых файлах при определении конца строки следует ориентироваться на символ `\n`, а двоичные файлы всегда должны открываться в двоичном режиме, чтобы предотвратить преобразование символов конца строки и кодирование/декодирование символов Юникода. Вообще тип содержимого файла определяется режимом его открытия, а режимы открытия определяют способы обработки содержимого, что в точности соответствует нашим желаниям.

Однако следует понимать, что в особых случаях может потребоваться открывать текстовые файлы в двоичном режиме. Так, в примерах из главы 6 мы иногда будем использовать двоичный режим для текстовых файлов, чтобы избежать возможных ошибок декодирования в Юникод при работе с файлами, созданными на разных платформах, где могут использоваться различные кодировки. Данный прием позволяет избежать ошибок кодирования, но при этом некоторые операции с текстом могут выполняться не так, как предполагается, – поиск в таком двоичном тексте не всегда может давать точные результаты, потому что искомый ключ также придется преобразовать в строку байтов в соответствии с некоторой кодировкой, возможно, несовместимой с кодировкой текста в файле.

В примере текстового редактора PyEdit, в главе 11, нам также потребуется перехватывать исключения, вызванные ошибками преобразования Юникода в утилите поиска «grep» по файлам в каталоге, и мы пойдем еще дальше, позволив пользователю определять кодировку символов содержимого файлов для целого дерева каталогов. Кроме того, когда необходимо явно выполнить преобразование символов конца строки в соответствии с соглашениями для двух разных платформ, может потребоваться прочитать текст в двоичном режиме, чтобы сохранить оригинальное представление концов строк, – при открытии в текстовом режиме они могут оказаться преобразованными в `\n` к моменту, когда данные попадут в сценарий.

Имеется также возможность запретить преобразование символов конца строки в текстовом режиме с помощью дополнительных аргументов функции `open`, которые мы не будем рассматривать здесь. Подробности ищите в описании аргумента `newline` в справочной документации по

функции open, но, в двух словах: если в этом аргументе передать пустую строку, это предотвратит преобразование символов конца строки и сохранит остальные особенности поведения текстового режима. Теперь обратимся к следующим двум типичным случаям использования двоичных файлов: работа с упакованными двоичными данными и произвольный доступ.

Работа с упакованными двоичными данными с помощью модуля struct

Используя символ *b* в аргументе режима функции open, вы получаете возможность открывать двоичные файлы с данными платформонезависимым способом, а также читать и записывать их содержимое с помощью обычных методов объекта файла. Но как обрабатывать двоичные данные после того, как они будут прочитаны? Эти данные будут возвращены сценарию в виде простой строки байтов, большая часть из которых наверняка будет соответствовать непечатаемым символам.

Если нужно лишь переписать двоичные данные в другой файл или передать другой программе, тогда все просто – достаточно записать строку байтов в другой файл, открытый в двоичном режиме. Если потребуется извлечь некоторое количество байтов из определенной позиции, можно воспользоваться операцией извлечения среза строки. При необходимости можно даже использовать битовые операции. Кроме того, имеется мощный инструмент, позволяющий получить двоичные данные в структурированном виде или сконструировать их, – модуль struct из стандартной библиотеки.

В модуле struct имеются функции для упаковывания и распаковывания двоичных данных, как если бы данные были созданы с помощью объявления struct языка C. Имеется возможность при упаковывании и распаковывании данных учитывать прямой или обратный порядок следования байтов (порядок следования байтов определяет, где будут находиться старшие значимые биты в двоичном представлении чисел, – слева или справа). Создание двоичного файла с данными, например, – достаточно простая задача: нужно упаковать значения языка Python в строку байтов и записать ее в файл. Строка формата в вызове pack ниже определяет: прямой порядок следования байтов (>), целое число, 4-символьную строку, короткое целое число и вещественное число:

```
>>> import struct
>>> data = struct.pack('>i4shf', 2, 'spam', 3, 1.234)
>>> data
b'\x00\x00\x00\x02spam\x00\x03?\x9d\xf3\xb6'
>>> file = open('data.bin', 'wb')
>>> file.write(data)
14
>>> file.close()
```

Обратите внимание, что модуль `struct` возвращает строку байтов: сейчас мы находимся в царстве двоичных данных, а не текста, и для сохранения должны использовать двоичные файлы. Как обычно, интерпретатор отображает большую часть байтов с упакованными двоичными данными, которые не соответствуют печатаемым символам, в виде шестнадцатеричных экранированных последовательностей `\xNN`. Чтобы выполнить обратное преобразование этих данных, нужно прочитать их из файла и передать модулю `struct` с той же строкой формата, как и при создании, – в результате получится кортеж значений, полученных в результате анализа строки байтов и преобразованных в объекты языка Python:

```
>>> import struct
>>> file = open('data.bin', 'rb')
>>> data = file.read()
>>> values = struct.unpack('>i4shf', data)
>>> values
(2, b'spam', 3, 1.2339999675750732)
```

Анализируемые строки – это также строки байтов, и к ним допускается применять строковые и битовые операции для более глубокого анализа:

```
>>> bin(values[0] | 0b1)       # доступ к битам и байтам
'0b11'
>>> values[1], list(values[1]), values[1][0]
(b'spam', [115, 112, 97, 109], 115)
```

Обратите также внимание, что здесь может пригодиться операция извлечения среза. 4-символьную строку из середины только что прочитанных упакованных двоичных данных легко получить, используя операцию извлечения среза. Числовые значения также можно извлекать подобным способом и передавать функции `struct.unpack` для преобразования:

```
>>> data
b'\x00\x00\x00\x02spam\x00\x03?\x9d\xf3\xb6'
>>> data[4:8]
b'spam'
>>> number = data[8:10]
>>> number
b'\x00\x03'
>>> struct.unpack('>h', number)
(3,)
```

Упакованные двоичные данные бывают получены из самых разных контекстов, включая некоторые виды сетевых взаимодействий и представление данных другими языками программирования. Однако все это не относится к разряду повседневных задач программирования, поэтому оставим описание подробностей за разделом с описанием модуля `struct` в руководстве по стандартной библиотеке Python.

Произвольный доступ к данным в файлах

При работе с двоичными файлами часто также применяется операция произвольного доступа. Ранее упоминалось, что добавление символа + в строку режима открытия файла позволяет выполнять обе операции, чтения и записи. Этот режим обычно используется вместе с методом seek объектов файлов, позволяющим выполнять чтение/запись произвольных участков файла. Такие гибкие режимы обработки файлов позволяют читать байты из одного места, записывать в другое и так далее. При объединении этих режимов с двоичным режимом появляется возможность извлекать и изменять произвольные байты в файле.

Выше для перехода в начало файла вместо операций закрытия файла и повторного его открытия использовался метод seek. Как уже упоминалось, операции чтения и записи всегда выполняются в текущей позиции в файле. При открытии файлов текущая позиция обычно устанавливается в смещение 0 от начала файла и перемещается вперед по мере чтения/записи данных. Метод seek позволяет переместить текущую позицию для следующей операции чтения/записи в другое место, для чего ему достаточно передать величину смещения в байтах.

Метод seek в языке Python принимает также второй необязательный аргумент, который определяет физический смысл первого аргумента и может принимать одно из трех значений: 0 – абсолютная позиция в файле (по умолчанию), 1 – смещение относительно текущей позиции и 2 – смещение относительно конца файла. Когда методу seek передается только аргумент смещения 0, это соответствует операции *перемотки файла в начало* (*rewind*): текущая позиция перемещается в начало файла. Вообще, метод seek поддерживает произвольный доступ на уровне смещения в байтах. Используя в качестве множителя размер записи в двоичном файле, можно организовать доступ к записям по их относительным позициям.

Метод seek можно использовать и без спецификатора + в строке режима для функции open (например, чтобы просто обеспечить произвольное чтение данных), но наибольшая гибкость достигается при работе с файлами, открытыми для чтения и записи. Возможность произвольного доступа поддерживается и для файлов, открытых в *текстовом режиме*. Но выполняющиеся в текстовом режиме операции кодирования/декодирования Юникода и преобразования символов конца строки сильно осложняют вычисление абсолютных смещений в байтах и длин, необходимых методам позиционирования и чтения, – представление ваших данных может значительно измениться при сохранении в файл. Кроме того, применение текстового режима может также ухудшить переносимость данных между платформами, где по умолчанию используются различные кодировки, если только вы не предполагаете всегда явно указывать кодировку файлов. Метод seek лучше подходит для работы с двоичными файлами; исключение составляет простой некодируемый текст ASCII, в котором отсутствуют символы конца строки.

Для демонстрации создадим файл в режиме "w+b" (эквивалент режима "wb+") и запишем в него некоторые данные – этот режим позволяет читать из файла и писать в него и создает новый пустой файл, если он существовал прежде (это относится ко всем режимам "w"). После записи данных мы вернемся в начало файла и прочитаем его содержимое (несколько целочисленных значений, возвращаемых вызовами методов в этом примере, было опущено ради экономии места):

```
>>> records = [bytes([char] * 8) for char in b'spam']
>>> records
[b'ssssssss', b'pppppppp', b'aaaaaaaa', b'mmmmmmmm']

>>> file = open('random.bin', 'w+b')
>>> for rec in records:                # записать четыре записи
...     size = file.write(rec)         # bytes означает двоичный режим
...
>>> file.flush()
>>> pos = file.seek(0)                 # прочитать файл целиком
>>> print(file.read())
b'ssssssssppppppppaaaaaaaammmmmmmm'
```

Теперь повторно откроем файл в режиме "r+b" – он также позволяет читать из файла и писать в него, но не очищает файл при открытии. На этот раз мы будем выполнять позиционирование и чтение с учетом размеров элементов данных («записей»), чтобы показать возможность получения и изменения записей в произвольном порядке:

```
c:\temp> python
>>> file = open('random.bin', 'r+b')
>>> print(file.read())                 # прочитать файл целиком
b'ssssssssppppppppaaaaaaaammmmmmmm'

>>> record = b'X' * 8
>>> file.seek(0)                       # изменить первую запись
>>> file.write(record)
>>> file.seek(len(record) * 2)         # изменить третью запись
>>> file.write(b'Y' * 8)

>>> file.seek(8)
>>> file.read(len(record))             # извлечь вторую запись
b'pppppppp'
>>> file.read(len(record))             # извлечь следующую (третью) запись
b'YYYYYYYY'

>>> file.seek(0)                       # прочитать файл целиком
>>> file.read()
b'XXXXXXXXppppppppYYYYYYYYmmmmmmmm'

c:\temp> type random.bin               # посмотреть файл за пределами Python
XXXXXXXXppppppppYYYYYYYYmmmmmmmm
```

Наконец, имейте в виду, что метод seek можно использовать, даже если файл открыт только для чтения. Следующий пример демонстрирует возможность чтения произвольных записей фиксированной длины. Обратите внимание, что при этом используется текстовый режим "r": поскольку данные представляют собой простой текст ASCII, где каждый символ представлен одним байтом, и текст не содержит символов конца строки, на данной платформе текстовый и двоичный режимы действуют одинаково:

```
c:\temp> python
>>> file = open('random.bin', 'r')  # текстовый режим можно использовать, если
                                    # не выполняется кодирование и отсутствуют
                                    # символы конца строки
>>> reclen = 8
>>> file.seek(reclen * 3)           # извлечь четвертую запись
>>> file.read(reclen)
'mmmmmmmm'
>>> file.seek(reclen * 1)           # извлечь вторую запись
>>> file.read(reclen)
'ppppppp'

>>> file = open('random.bin', 'rb') # в данном случае двоичный режим действует
                                    # точно так же
>>> file.seek(reclen * 2)           # извлечь третью запись
>>> file.read(reclen)               # вернет строку байтов
b'YYYYYYYY'
```

Но в общем случае текстовый режим не следует использовать, если вам требуется произвольный доступ к записям (за исключением файлов с простым некодируемым текстом, подобным ASCII, не содержащим символов конца строки). Символы конца строки могут преобразовываться в Windows, а применение кодировок Юникода может вносить различные искажения – оба эти преобразования существенно осложняют возможность позиционирования по абсолютному смещению. Например, в следующем фрагменте соответствие между строкой Python и ее кодированным представлением в файле нарушается сразу же за первым не-ASCII символом:

```
>>> data = 'sp\xe4m'                                  # данные в сценарии
>>> data, len(data)                                   # 4 символа Юникода,
('späm', 4)                                           # 1 символ не-ASCII
>>> data.encode('utf8'), len(data.encode('utf8'))     # байты для записи в файл
(b'sp\xc3\xa4m', 5)

>>> f = open('test', mode='w+', encoding='utf8')      # текст. режим, кодирование
>>> f.write(data)
>>> f.flush()
>>> f.seek(0); f.read(1)                              # работает для байтов ascii
's'
>>> f.seek(2); f.read(1)                              # 2-байтовый не-ASCII
'ä'
```

```
>>> data[3]                                    # а в смещении 3 - не 'm' !
'm'
>>> f.seek(3); f.read(1)
UnicodeDecodeError: 'utf8' codec can't decode byte 0xa4 in position 0:
unexpected code byte
```

(UnicodeDecodeError: кодек 'utf8' не может преобразовать байт 0xa4 в позиции 0: неопознанный код)

Как видите, режимы открытия файлов в Python обеспечивают необходимую гибкость при работе с файлами в программах. А модуль os предлагает еще более широкие возможности для обработки файлов, которые представлены в следующем разделе.

Низкоуровневые инструменты в модуле os для работы с файлами

Модуль os содержит дополнительный набор функций для работы с файлами, отличных от инструментов, которыми располагают встроенные *объекты* файлов, демонстрировавшиеся в предыдущих примерах. Например, ниже приводится неполный список функций в модуле os, имеющих отношение к файлам:

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

Открывает файл, возвращает его дескриптор

os.read(*descriptor*, *N*)

Читает не более *N* байтов и возвращает строку байтов

os.write(*descriptor*, *string*)

Записывает в файл байты из строки байтов *string*

os.lseek(*descriptor*, *position*, *how*)

Перемещается в позицию *position* в файле

С технической точки зрения, функции из модуля os обрабатывают файлы по их *дескрипторам*, которые представляют собой целочисленные коды или «описатели» (handles), идентифицирующие файлы в операционной системе. Файлы, представленные дескрипторами, интерпретируются как обычные двоичные файлы, к которым не применяются ни преобразование символов конца строки, ни кодирование текста, о которых рассказывалось в предыдущем разделе. Фактически, за исключением отдельных особенностей, таких как буферизация, операции с файлами, представленными дескрипторами, мало чем отличаются от операций, поддерживаемых объектами файлов для двоичного режима. При работе с такими файлами мы также читаем и пишем строки типа bytes, а не str. Однако так как инструменты для работы с файлами с использованием дескрипторов, представленные в модуле os, – более низкого уровня и более сложны в применении, чем встроенные объекты файлов, создаваемые с помощью встроенной функции open, то следует

использовать последние во всех ситуациях, за исключением отдельных случаев специальной обработки файлов.[1]

Использование файлов, возвращаемых os.open

Чтобы дать вам общее представление об этом наборе инструментов, проведем несколько интерактивных экспериментов. Встроенные объекты файлов и файловые дескрипторы модуля os обрабатываются различными наборами инструментов, но в реальности они связаны между собой – объекты файлов просто добавляют дополнительную логику поверх дескрипторов файлов.

Метод fileno объекта файла возвращает целочисленный дескриптор, ассоциированный со встроенным объектом файла. Например, объекты файлов стандартных потоков ввода-вывода имеют дескрипторы 0, 1 и 2; вызов функции os.write для отправки данных в stdout по дескриптору дает тот же эффект, что и 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')    # записать с помощью метода
Hello stdio world                              # объекта файла
18
>>> import os
>>> os.write(1, b'Hello descriptor world\n')   # записать с помощью модуля os
Hello descriptor world
23
```

Поскольку объекты файлов, открываемые явно, ведут себя точно так же, с одинаковым успехом для обработки конкретного внешнего файла на компьютере можно использовать встроенную функцию open, инструменты из модуля os или и то и другое вместе:

```
>>> file = open(r'C:\temp\spam.txt', 'w')  # создать внешний файл, объект
>>> file.write('Hello stdio file\n')       # записать с помощью объекта файла
>>> file.flush()                           # или сразу - функции os.write
>>> fd = file.fileno()                     # получить дескриптор из объекта
```

[1] Например, для обработки *каналов*, описываемых в главе 5. В Python функция os.pipe возвращает два файловых дескриптора, которые можно обрабатывать средствами модуля os или обертывать их объектами файлов с помощью функции os.fdopen. При использовании файловых инструментов из модуля os данные через каналы передаются в виде строк байтов, а не текста. Кроме того, низкоуровневые механизмы могут также потребоваться для работы с некоторыми файлами устройств.

```
>>> fd
3
>>> import os
>>> os.write(fd, b'Hello descriptor file\n') # записать с помощью модуля os
>>> file.close()

C:\temp> type spam.txt                    # строки, записанные
Hello stdio file                          # двумя способами
Hello descriptor file
```

Флаги режима os.open

Зачем же нужны дополнительные файловые средства в модуле os? Если вкратце, то они обеспечивают более низкоуровневое управление обработкой файлов. Встроенная функция open проста в использовании, но она ограничена возможностями файловой системы, которую использует, и добавляет некоторые дополнительные особенности, которые могут быть нежелательны. Модуль os позволяет сценариям быть более точными; например, следующий фрагмент открывает дескриптор файла в двоичном режиме для чтения-записи, выполняя битовую операцию «ИЛИ» над двумя флагами режима, экспортируемыми модулем os:

```
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> os.read(fdfile, 20)
b'Hello stdio file\r\nHe'

>>> os.lseek(fdfile, 0, 0)     # вернуться в начало файла
>>> os.read(fdfile, 100)       # в двоичном режиме сохраняются "\r\n"
b'Hello stdio file\r\nHello descriptor file\n'

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

C:\temp> type spam.txt
HELLO stdio file
Hello descriptor file
```

В данном случае эквивалентный режим открытия с помощью встроенной функции open определяется строками "rb+" и "r+b":

```
>>> file = open(r'C:\temp\spam.txt', 'rb+')  # то же самое, но с помощью open
>>> file.read(20)                            # и объектов файлов
b'HELLO stdio file\r\nHe'
>>> file.seek(0)
>>> file.read(100)
b'HELLO stdio file\r\nHello descriptor file\n'
>>> file.seek(0)
>>> file.write(b'Jello')
5
>>> file.seek(0)
```

```
>>> file.read()
b'Jello stdio file\r\nHello descriptor file\n'
```

В некоторых системах флаги для функции `os.open` позволяют указывать более сложные режимы – например, *исключительный доступ* (`O_EXCL`) и *неблокирующий* режим (`O_NONBLOCK`). Некоторые из этих флагов не переносимы между платформами (еще одна причина в пользу встроенных объектов файлов). Найти полный список других флагов открытия можно в руководстве по библиотеке или вызвав на своем компьютере функцию `dir(os)`.

И последнее замечание: в Python использование функции `os.open` с флагом `O_EXCL` на сегодняшний день является наиболее переносимым способом исключить возможность параллельного изменения файла или обеспечить синхронизацию с другими процессами. Где может использоваться эта особенность, мы увидим в следующей главе, когда приступим к исследованию инструментов параллельной обработки данных. Программам, параллельно выполняющимся на сервере, к примеру, может потребоваться устанавливать блокировку на файлы, прежде чем изменять их, если подобные изменения могут одновременно запрашиваться несколькими потоками выполнения или процессами.

Обертывание дескрипторов объектами файлов

Ранее было показано, как перейти от использования объекта файла к использованию дескриптора с помощью метода объекта файла `fileno`, – получив дескриптор, мы можем использовать инструменты из модуля `os` для выполнения низкоуровневых операций с файлом. Но можно пойти и обратным путем – функция `os.fdopen` обертывает дескриптор файла объектом файла. Поскольку преобразования могут выполняться в обоих направлениях, мы можем выбирать любой набор инструментов – объект файла или модуль `os`:

```
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> fdfile
3
>>> objfile = os.fdopen(fdfile, 'rb')
>>> objfile.read()
b'Jello stdio file\r\nHello descriptor file\n'
```

Фактически мы можем обернуть дескриптор файла любым объектом файла, открытым в текстовом или в двоичном режиме. В текстовом режиме операции чтения и записи будут производить кодирование/декодирование Юникода и преобразование символов конца строки, с которыми мы познакомились выше, и для работы с ними необходимо будет использовать строки типа `str`, а не `bytes`:

```
C:\...\PP4E\System> python
>>> import os
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> objfile = os.fdopen(fdfile, 'r')
```

```
>>> objfile.read()
'Jello stdio file\nHello descriptor file\n'
```

Встроенная функция open в Python 3.X также может принимать дескриптор файла вместо строки с его именем. В этом режиме она действует практически так же, как функция os.fdopen, но обеспечивает более полный контроль. Например, можно использовать дополнительные аргументы, чтобы определить кодировку для текста и подавить операцию закрытия дескриптора, которая выполняется по умолчанию. Однако на практике функция os.fdopen в версии 3.X принимает те же дополнительные аргументы, потому что она была переопределена и теперь вызывает встроенную функцию open (смотрите файл *os.py* в стандартной библиотеке):

```
C:\...\PP4E\System> python
>>> import os
>>> fdfile = os.open(r'C:\temp\spam.txt', (os.O_RDWR | os.O_BINARY))
>>> fdfile
3

>>> objfile = open(fdfile, 'r', encoding='latin1', closefd=False)
>>> objfile.read()
'Jello stdio file\nHello descriptor file\n'

>>> objfile = os.fdopen(fdfile, 'r', encoding='latin1', closefd=True)
>>> objfile.seek(0)
>>> objfile.read()
'Jello stdio file\nHello descriptor file\n'
```

Далее в книге мы будем использовать этот прием обертывания в объекты файлов, чтобы упростить работу в текстовом режиме с каналами и другими объектами на основе дескрипторов (например, сокеты обладают методом makefile, позволяющим добиться похожего эффекта).

Другие инструменты для работы с файлами в модуле os

В модуле os имеется также ряд инструментов для работы с файлами, которые принимают строку пути к файлу и выполняют ряд операций, связанных с файлами, 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 инструментов:

```
>>> os.chmod('spam.txt', 0o777)        # разрешить доступ всем пользователям
```

Функции os.chmod установки прав доступа к файлу передается строка из девяти битов, состоящая из трех групп, по три бита в каждой. Эти три группы определяют права доступа, слева направо, для пользователя-владельца файла, для группы пользователей, которой принадлежит файл, и для всех остальных. Три бита внутри каждой группы отражают право на чтение, на запись и на выполнение. Если какой-то бит в этой

строке равен «1», это означает разрешение на выполнение соответствующей операции. Например, восьмеричное число 0777 является строкой из девяти единичных битов в двоичном представлении и разрешает все три вида доступа для всех трех групп пользователей; восьмеричное число 0600 означает возможность только чтения и записи для пользователя, который владеет файлом (восьмеричное число 0600 в двоичной записи дает 110 000 000).

Эта схема ведет свое происхождение от системы прав доступа в Unix, но работает также в Windows. Если она вас озадачила, посмотрите описание команды *chmod* в документации по вашей системе (например, в страницах руководства Unix). Идем дальше:

```
>>> os.rename(r'C:\temp\spam.txt', r'C:\temp\eggs.txt') # откуда, куда

>>> os.remove(r'C:\temp\spam.txt')                     # удалить файл?
WindowsError: [Error 2] The system cannot find the file specified: 'C:\\
temp\\...'
```

(WindowsError: [Error 2] Системе не удается найти указанный путь: 'C:\\ temp\\...')

```
>>> os.remove(r'C:\temp\eggs.txt')
```

Использованная здесь функция `os.rename` изменяет имя файла; функция `os.remove` удаляет файл, она синонимична функции `os.unlink` (имя последней – имя, которое имеет эта функция в Unix, но оно не знакомо пользователям других платформ)[1]. Модуль `os` также экспортирует системный вызов `stat`:

```
>>> open('spam.txt', 'w').write('Hello stat world\n') # +1 для символа \r
17
>>> import os
>>> info = os.stat(r'C:\temp\spam.txt')
>>> info
nt.stat_result(st_mode=33206, st_ino=0, st_dev=0, st_nlink=0, st_uid=0,
st_gid=0, st_size=18, st_atime=1267645806, st_mtime=1267646072, st_
ctime=1267645806)

>>> info.st_mode, info.st_size  # через атрибуты именованного кортежа
(33206, 18)

>>> import stat
>>> info[stat.ST_MODE], info[stat.ST_SIZE] # через константы в модуле stat
```

[1] Похожие инструменты вы найдете также в модуле `shutil`, в стандартной библиотеке Python. Он содержит высокоуровневые инструменты для выполнения операций копирования и удаления файлов и многие другие. В главе 6 мы дополнительно напишем инструменты, позволяющие выполнять операции над каталогами, такие как сравнение, копирование и поиск, после того как познакомимся с инструментами для работы с каталогами далее в этой главе.

```
(33206, 18)
>>> stat.S_ISDIR(info.st_mode), stat.S_ISREG(info.st_mode)
(False, True)
```

Функция os.stat возвращает кортеж величин (в версии 3.X это особая разновидность кортежа, элементы которого имеют имена), представляющих низкоуровневую информацию о файле с указанным именем, а модуль stat экспортирует константы и функции для получения этой информации переносимым способом. Например, значение, получаемое из результата функции os.stat по индексу stat.ST_SIZE, соответствует размеру файла, а вызов функции stat.S_ISDIR с параметром «режим», полученным из результата функции os.stat, позволяет проверить, является ли файл каталогом. Однако, как было показано выше, обе эти операции доступны и в модуле os.path, поэтому на практике редко возникает необходимость использовать функцию os.stat; исключение составляют низкоуровневые запросы:

```
>>> path = r'C:\temp\spam.txt'
>>> os.path.isdir(path), os.path.isfile(path), os.path.getsize(path)
(False, True, 18)
```

Сканеры файлов

Прежде чем закончить обзор инструментов для работы с файлами, реализуем более практичную задачу и проиллюстрируем кое-что из того, что мы уже видели. В отличие от некоторых языков командной оболочки, в Python нет неявной процедуры циклического сканирования файла, но написать такую универсальную процедуру, пригодную для многократного использования, несложно. Модуль в примере 4.1 определяет универсальную процедуру сканирования файлов, которая просто применяет переданную в нее функцию к каждой строке внешнего файла.

Пример 4.1. PP4E\System\Filetools\scanfile.py

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

Функции scanner безразлично, какая функция обработки строк в нее передана, чем и определяется ее универсальность: она готова применить *любую* функцию одного аргумента, уже существующую или которая может появиться в будущем, ко всем строкам в текстовом файле. Если реализацию этого модуля поместить в каталог, входящий в путь поиска модулей, им можно будет воспользоваться всякий раз, когда потребуется выполнить построчный обход файл. В примере 4.2 приводится клиентский сценарий, выполняющий простое преобразование строк.

Пример 4.2. PP4E\System\Filetools\commands.py

```
#!/usr/local/bin/python
from sys import argv
from scanfile import scanner
class UnknownCommand(Exception): pass

def processLine(line):                  # определить функцию,
    if line[0] == '*':                  # применяемую к каждой строке
        print("Ms.", line[1:-1])
    elif line[0] == '+':
        print("Mr.", line[1:-1])        # отбросить первый и последний символы
    else:
        raise UnknownCommand(line)      # возбудить исключение

filename = 'data.txt'
if len(argv) == 2: filename = argv[1]  # аргумент командной строки с именем
scanner(filename, processLine)         # файла запускает сканер
```

Для текстового файла *hillbillies.txt*:

```
*Granny
+Jethro
*Elly May
+"Uncle Jed"
```

наш сценарий commands.py вернет следующие результаты:

```
C:\...\PP4E\System\Filetools> python commands.py hillbillies.txt
Ms. Granny
Mr. Jethro
Ms. Elly May
Mr. "Uncle Jed"
```

Все работает, тем не менее существует множество альтернативных способов реализации обоих примеров, и какие-то из них могут предлагать более удачные решения. Например, процессор команд, представленный в примере 4.2, можно было бы реализовать, как показано ниже. Преимущества этой реализации становятся более очевидными с ростом обрабатываемых вариантов – управляемый данными подход может оказаться короче и проще в сопровождении, чем длинная инструкция `if` с избыточными, по сути, действиями (если вам когда-нибудь потребуется изменить способ вывода строк, в следующей реализации вы сможете сделать это, изменив всего одну строку):

```
commands = {'*': 'Ms.', '+': 'Mr.'} # данные изменять проще, чем код?

def processLine(line):
    try:
        print(commands[line[0]], line[1:-1])
    except KeyError:
        raise UnknownCommand(line)
```

Сканер также можно было бы улучшить. Как правило, перемещение обработки из программного кода Python во встроенные инструменты приводит к увеличению скорости. Например, если скорость имеет большое значение, сканер файлов можно было бы сделать быстрее, заменив в примере 4.1 вызов функции `readline` *итератором объекта файла* (в эффективности которого вы уже имели возможность убедиться):

```
def scanner(name, function):
    for line in open(name, 'r'):   # построчное сканирование
        function(line)             # вызов объекта функции
```

Еще больших чудес в примере 4.1 можно достичь с помощью таких инструментов итераций, как встроенная функция `map`, генераторы списков и выражения-генераторы. Ниже приводится минималистская версия. Цикл `for` замещается вызовом функции `map` или генератором, и Python сам закрывает файл на этапе сборки мусора или при выходе из сценария (в процессе обработки во всех реализациях создается список результатов, однако такое неэкономное расходование ресурсов вполне допустимо, за исключением очень больших файлов):

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

def scanner(name, function):
    [function(line) for line in open(name, 'r')]

def scanner(name, function):
    list(function(line) for line in open(name, 'r'))
```

Фильтры файлов

Предыдущий пример работает, как предполагалось, но как быть, если во время сканирования файла нам потребуется файл *изменить*? В примере 4.3 показаны два подхода: в одном используются явные файлы, а в другом стандартные потоки ввода-вывода, которые можно перенаправить в командной строке.

Пример 4.3. PP4E\System\Filetools\filters.py

```
import sys

def filter_files(name, function):      # фильтрация файлов через функцию
    input = open(name, 'r')            # создать объекты файлов
    output = open(name + '.out', 'w')  # выходной файл
    for line in input:
        output.write(function(line))   # записать измененную строку
    input.close()
    output.close()                     # выходной файл имеет расширение '.out'

def filter_stream(function):           # отсутствуют явные файлы
    while True:                        # использовать стандартные потоки
        line = sys.stdin.readline()    # или: input()
```

```
        if not line: break
        print(function(line), end='')  # или: sys.stdout.write()

if __name__ == '__main__':
    filter_stream(lambda line: line)   # копировать stdin в stdout, если
                                       # запущен как самостоятельный сценарий
```

Обратите внимание, что применение такой новейшей особенности, как *менеджеры контекста*, обсуждавшейся выше, позволило бы сэкономить несколько строк программного кода в реализации фильтра из примера 4.3, опирающегося на использование файлов, и гарантировало бы немедленное закрытие файлов в случае появления исключения в функции обработки:

```
def filter_files(name, function):
    with open(name, 'r') as input, open(name + '.out', 'w') as output:
        for line in input:
            output.write(function(line)) # записать измененную строку
```

И снова, применение *итераторов объектов файлов* позволило бы упростить реализацию фильтра на основе потоков ввода-вывода:

```
def filter_stream(function):
    for line in sys.stdin:         # автоматически выполняет построчное чтение
        print(function(line), end='')
```

Поскольку стандартные потоки ввода-вывода открываются автоматически, они обычно проще в использовании. Если запустить этот пример, как самостоятельный сценарий, он просто скопирует `stdin` в `stdout`:

```
C:\...\PP4E\System\Filetools> filters.py < hillbillies.txt
*Granny
+Jethro
*Elly May
+"Uncle Jed"
```

Однако этот модуль более полезен, когда он импортируется как библиотека (клиент предоставляет функцию обработки строк):

```
>>> from filters import filter_files
>>> filter_files('hillbillies.txt', str.upper)
>>> print(open('hillbillies.txt.out').read())
*GRANNY
+JETHRO
*ELLY MAY
+"UNCLE JED"
```

В оставшейся части книги мы часто будем видеть примеры использования файлов, особенно в наиболее полных и практичных примерах системных программ в главе 6. Однако сначала познакомимся с инструментами обработки жилища наших файлов.

Инструменты для работы с каталогами

Одной из наиболее частых задач для утилит командной оболочки является применение операций к множеству файлов, находящихся в *каталоге* – «папке» на языке Windows. Сценарии, способные выполнять операции над группой файлов, позволяют автоматизировать (то есть *программировать*) задачи, которые в противном случае пришлось бы многократно повторять вручную.

Например, допустим, что нужно найти во всех файлах с программным кодом Python из каталога разработки имя глобальной переменной (вы могли забыть, где оно используется). Для каждой платформы существует множество способов решить эту задачу (например, команды `find` и `grep` в Unix), но сценарии Python, выполняющие такие задачи, будут работать на любой платформе, где работает Python, – в Windows, Unix, Linux, Macintosh и практически на любой другой распространенной платформе. Достаточно просто скопировать сценарий на любой компьютер, где предполагается его использовать, и он будет работать независимо от имеющихся на нем утилит, – для этого необходимо иметь лишь интерпретатор Python. Кроме того, программирование таких задач на языке Python позволяет по ходу дела выполнять любые действия – замену, удаление и любые другие, какие только можно реализовать на языке Python.

Обход одного каталога

Чаще всего при написании таких инструментов сначала получают список имен файлов, которые нужно обработать, а затем пошагово обходят его в цикле `for`, поочередно обрабатывая каждый файл. Весь фокус состоит в том, чтобы научиться получать в сценариях такой список содержимого каталога. Существует по меньшей мере три способа сделать это: выполнить команды оболочки для получения списка с помощью `os.popen`, отыскать 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 получения списка содержимого каталога с помощью os.popen

Скажите-ка, как вы получали списки файлов в каталоге до того, как услышали о Python? Если у вас нет опыта работы с инструментами командной строки, ответ может быть следующим: «Ну, я запускал в Windows проводник и щелкал, куда нужно». Но здесь у нас речь идет о механизмах, менее ориентированных на графический интерфейс, то есть о механизмах командной строки.

Для получения списков файлов в Unix обычно используется команда `ls`; в Windows списки можно создавать вводом `dir` в окне консоли MS-DOS.

Поскольку сценарии Python могут выполнить любую команду оболочки с помощью os.popen, они являются самым универсальным способом получения содержимого каталога из программ на языке Python. Мы уже встречались с функцией os.popen в предыдущей главе – она выполняет команду оболочки и возвращает объект файла, из которого можно прочесть вывод команды. Для иллюстрации допустим сначала, что имеется следующая структура каталогов – на моем ноутбуке с Windows есть обе команды, dir и Unix-подобная ls из Cygwin:

```
c:\temp> dir /B
parts
PP3E
random.bin
spam.txt
temp.bin
temp.txt

c:\temp> c:\cygwin\bin\ls
PP3E parts random.bin spam.txt temp.bin temp.txt

c:\temp> c:\cygwin\bin\ls parts
part0001 part0002 part0003 part0004
```

Имена *parts* и *PP3E* являются здесь подкаталогами, вложенным в каталог *C:\temp* (последний из них является копией дерева каталогов с примерами для предыдущего издания книги, часть из которых я использовал в этом издании). Теперь мы знаем, что сценарии могут получать списки имен файлов и каталогов на этом уровне, просто запуская специфическую для платформы команду и читая полученный вывод (текст, обычно выводимый в окно консоли):

```
C:\temp> python
>>> import os
>>> os.popen('dir /B').readlines()
['parts\n', 'PP3E\n', 'random.bin\n', 'spam.txt\n', 'temp.bin\n', 'temp.txt\n']
```

Строки, возвращаемые командой оболочки, содержат замыкающий символ конца строки, но его легко можно отсечь. Кроме того, функция os.popen возвращает итератор, точно такой же, как итератор объектов файлов:

```
>>> for line in os.popen('dir /B'):
...     print(line[:-1])
...
parts
PP3E
random.bin
spam.txt
temp.bin
temp.txt
```

```
>>> lines = [line[:-1] for line in os.popen('dir /B')]
>>> lines
['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']
```

В случае объектов каналов действие итераторов может иметь еще более значимый эффект, чем просто уход от одновременной загрузки всех результатов в память: метод `readlines` всегда блокирует вызывающий процесс, пока не завершится порожденная программа, тогда как при использовании итераторов этого не происходит.

Обе команды, `dir` и `ls`, позволяют задавать требуемые образцы имен файлов и имен каталогов, список содержимого которых должен быть получен, с помощью шаблонов имен. В следующем примере мы снова просто выполняем команды оболочки, поэтому годится все, что можно ввести в командной строке:

```
>>> os.popen('dir *.bin /B').readlines()
['random.bin\n', 'temp.bin\n']

>>> os.popen(r'c:\cygwin\bin\ls *.bin').readlines()
['random.bin\n', 'temp.bin\n']

>>> list(os.popen(r'dir parts /B'))
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']

>>> [fname for fname in os.popen(r'c:\cygwin\bin\ls parts')]
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']
```

Эти вызовы используют универсальные инструменты и все действуют, как было заявлено. Однако выше отмечалось, что недостатками `os.popen` являются необходимость использования команд оболочки, специфических для платформы, и потеря производительности при запуске независимых программ. На практике различные инструменты могут возвращать различные результаты:

```
>>> list(os.popen(r'dir parts\part* /B'))
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']
>>>
>>> list(os.popen(r'c:\cygwin\bin\ls parts/part*'))
['parts/part0001\n', 'parts/part0002\n', 'parts/part0003\n', 'parts/part0004\n']
```

Следующие два альтернативных приема проявляют себя лучше в обоих отношениях.

Модуль glob

Термин *globbing* (глобальный поиск по шаблону) происходит от группового символа `*`, используемого в шаблонах имен файлов. На компьютерном сленге символ `*` трактуется, как «glob» (группа символов). Более приземленно, глобальный поиск по шаблону просто означает получение имен всех элементов в каталоге – файлов и подкаталогов, имена

которых соответствуют заданному шаблону. В командных оболочках Unix при глобальном поиске шаблоны имен файлов, указанные в командной строке, расширяются до всех совпадающих имен еще перед выполнением команды. В Python можно делать нечто похожее, вызывая встроенную функцию `glob.glob`, – инструмент, принимающий шаблон имени файла и возвращающий список (не генератор) имен файлов, соответствующих этому шаблону:

```
>>> import glob
>>> glob.glob('*')
['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']

>>> glob.glob('*.bin')
['random.bin', 'temp.bin']

>>> glob.glob('parts')
['parts']

>>> glob.glob('parts/*')
['parts\\part0001', 'parts\\part0002', 'parts\\part0003', 'parts\\part0004']

>>> glob.glob('parts\part*')
['parts\\part0001', 'parts\\part0002', 'parts\\part0003', 'parts\\part0004']
```

Для определения шаблонов в функции `glob` используется обычный синтаксис шаблонов имен файлов, используемый в командных оболочках: `?` означает один любой символ, `*` означает любое число символов, а `[]` означает множество символов, доступных для выбора.[1] Если поиск нужно осуществлять в каталоге, отличном от текущего рабочего каталога, в шаблон нужно включить путь к каталогу. Кроме того, модуль принимает разделители имен каталогов в стиле Unix или DOS (/ или \). Эта функция реализована так, что не вызывает команды оболочки (она использует функцию `os.listdir`, описываемую в следующем разделе) и потому должна выполняться быстрее и лучше переноситься на все платформы Python, чем показанные выше приемы с применением функции `os.popen`.

Вообще функция `glob` несколько мощнее, чем здесь описано. Получение списка файлов в каталоге является лишь одной из ее возможностей поиска по шаблону. Например, ее можно использовать для получения списка имен из нескольких каталогов, так как каждый уровень в передаваемом пути к каталогу также можно определить в виде шаблона:

```
>>> for path in glob.glob(r'PP3E\Examples\PP3E\*\s*.py'): print(path)
...
PP3E\Examples\PP3E\Lang\summer-alt.py
```

[1] В действительности функция `glob` просто использует стандартный модуль `fnmatch` для поиска имен по шаблону; смотрите описание `fnmatch` в примере модуля `find` в главе 6.

```
PP3E\Examples\PP3E\Lang\summer.py
PP3E\Examples\PP3E\PyTools\search_all.py
```

Здесь мы получили список имен файлов, соответствующих шаблону `s*.py`, из двух разных каталогов. Так как в качестве имени предшествующего каталога был использован групповой символ `*`, Python перебрал все возможные пути к файлам. Запуская команды оболочки с помощью функции `os.popen`, такого же результата можно добиться, только если подобная возможность поддерживается самой командной оболочкой или командой вывода списка файлов.

Функция os.listdir

Функция `listdir` из модуля `os` является еще одним способом получить список имен файлов. Но она принимает не шаблон имени файла, а простую строку с именем каталога и возвращает список, содержащий имена всех файлов в каталоге – как просто файлов, так и вложенных подкаталогов, – для использования в вызывающем сценарии:

```
>>> import os
>>> os.listdir('.')
['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']
>>>
>>> os.listdir(os.curdir)
['parts', 'PP3E', 'random.bin', 'spam.txt', 'temp.bin', 'temp.txt']
>>>
>>> os.listdir('parts')
['part0001', 'part0002', 'part0003', 'part0004']
```

Эта функция также не привлекает к работе команды оболочки, и поэтому данный способ является не только быстрым, но и переносимым на все основные платформы Python. Результат функции не упорядочен никаким образом (но может быть отсортирован методом списков `sort` или функцией `sorted`); возвращает базовые имена файлов без путей к каталогам; не включает имена каталогов «`.`» или «`..`» и содержит имена файлов и подкаталогов для данного уровня.

Чтобы сравнить все три способа, запустим их друг за другом для явно заданного каталога. Они отличаются некоторыми деталями, но в целом являются вариациями на одну и ту же тему: функция `os.popen` возвращает символы конца строки и способна сортировать имена файлов на некоторых платформах, функция `glob.glob` принимает шаблоны и возвращает полные имена файлов с путями, а функция `os.listdir` принимает обычное имя каталога и возвращает имена файлов без путей к каталогам:

```
>>> os.popen('dir /b parts').readlines()
['part0001\n', 'part0002\n', 'part0003\n', 'part0004\n']

>>> glob.glob(r'parts\*')
['parts\\part0001', 'parts\\part0002', 'parts\\part0003', 'parts\\part0004']
```

```
>>> os.listdir('parts')
['part0001', 'part0002', 'part0003', 'part0004']
```

Из этих трех способов лучшими вариантами являются функции `glob` и `listdir`, если важна переносимость сценария и единообразие результатов, при этом функция `listdir` в последних версиях Python выглядит самой быстрой (тем не менее советую замеры производительности произвести самостоятельно – реализация может со временем измениться).

Разбиение и объединение результатов вывода

В предыдущем примере отмечалось, что функция `glob` возвращает полные имена файлов с путями, а функция `listdir` возвращает простые базовые имена файлов. В сценариях для удобства обработки часто требуется разбивать результаты функции `glob`, чтобы получить базовые имена, либо добавлять полные пути в результаты функции `listdir`. Такие преобразования легко реализуются, 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 с помощью функции `glob`, и затем добавить впереди них другие имена каталогов:

```
>>> dirname = r'C:\temp\parts'
>>>
>>> import glob
>>> for file in glob.glob(dirname + '/*'):
...     head, tail = os.path.split(file)
...     print(head, tail, '=>', ('C:\\Other\\' + tail))
...
C:\temp\parts part0001 => C:\Other\part0001
C:\temp\parts part0002 => C:\Other\part0002
C:\temp\parts part0003 => C:\Other\part0003
C:\temp\parts part0004 => C:\Other\part0004
```

Здесь после `=>` показаны полные имена файлов, которые получатся после перемещения. Напротив, сценарию, который должен обработать все файлы в каталоге, отличном от того, в котором он выполняется, вероятно, потребуется добавить к результатам функции `listdir` имя целевого каталога, прежде чем предавать имена файлов другим инструментам:

```
>>> import os
>>> for file in os.listdir(dirname):
...     print(dirname, file, '=>', os.path.join(dirname, file))
...
C:\temp\parts part0001 => C:\temp\parts\part0001
C:\temp\parts part0002 => C:\temp\parts\part0002
C:\temp\parts part0003 => C:\temp\parts\part0003
C:\temp\parts part0004 => C:\temp\parts\part0004
```

Когда вы начнете писать действующие инструменты для работы с каталогами, похожие на те, что мы будем разрабатывать в главе 6, пользование этими функциями войдет у вас в привычку.

Обход деревьев каталогов

Возможно, вы обратили внимание, что все предшествующие приемы в этом разделе возвращают имена файлов только *из одного* каталога (единственным исключением является глобальный поиск по шаблону). Такой подход годится в большинстве случаев, но что если потребуется применить операцию к каждому файлу в каждом каталоге и во всех подкаталогах *дерева* каталогов?

Например, допустим, что требуется найти в сценариях на языке Python все вхождения некоторого глобального имени. Однако на этот раз наши сценарии организованы в виде *пакета* модулей – каталога с вложенными подкаталогами, которые могут содержать собственные подкаталоги. Можно вручную запускать наш гипотетический поисковый механизм для одного каталога в каждом из подкаталогов в дереве, но это утомительно, чревато ошибками и точно не доставит удовольствия.

К счастью, реализовать обработку дерева каталогов на языке Python почти так же просто, как и просканировать единственный каталог. Можно написать рекурсивную процедуру обхода дерева или использовать утилиту перемещения по дереву, встроенную в модуль `os`. Такие инструменты можно использовать для поиска, копирования, сравнения и выполнения любых других операций над произвольными деревьями каталогов на любой платформе, где выполняется Python (то есть почти всюду).

Функция обхода дерева os.walk

Чтобы облегчить применение операции ко всем файлам в дереве каталогов, в составе Python поставляется утилита, выполняющая обход дерева и запускающая в каждом каталоге указанную функцию. Функции `os.walk` передается имя корневого каталога, и она автоматически обходит все дерево от корня и ниже.

По своей сути функция `os.walk` является *функцией-генератором* – для каждого каталога в дереве она возвращает кортеж из трех элементов, содержащий имя текущего каталога, а также списки всех файлов и всех подкаталогов в текущем каталоге. Так как это функция-генератор, обход дерева обычно реализуется с помощью цикла `for` (или другого инструмента итераций). В каждой итерации функция перемещается к следующему подкаталогу, а инструкция цикла выполняет свое тело для следующего уровня в дереве (например, открывает все файлы в этом подкаталоге и производит поиск по их содержимому).

На первый взгляд, такое описание может показаться ужасно сложным, но когда вы привыкнете к функции `os.walk`, все окажется довольно простым. В следующем фрагменте, например, тело цикла выполняется для каждого каталога в дереве с корнем в текущем рабочем каталоге. Цикл просто выводит имя каталога и имена всех файлов в нем, добавляя к ним имя каталога. Описать это на языке Python проще, чем на

обычном языке (перед тем как запускать этот пример, я удалил каталог PP3E, чтобы сократить вывод):

```
>>> import os
>>> for (dirname, subshere, fileshere) in os.walk('.'):
...     print('[' + dirname + ']')
...     for fname in fileshere:
...         print(os.path.join(dirname, fname)) # обработка одного файла
...
[.]
.\random.bin
.\spam.txt
.\temp.bin
.\temp.txt
[.\parts]
.\parts\part0001
.\parts\part0002
.\parts\part0003
.\parts\part0004
```

Иными словами, мы реализовали наш собственный, легко изменяемый инструмент рекурсивного вывода содержимого каталога на языке Python, Поскольку нам может потребоваться подправить его и использовать где-нибудь еще, давайте сделаем его постоянно доступным в виде файла модуля, как показано в примере 4.4, – теперь, когда мы проработали детали в интерактивном режиме.

Пример 4.4. PP4E\System\Filetools\lister_walk.py

```
"выводит список файлов в дереве каталогов с помощью os.walk"

import sys, os

def lister(root):                                    # для корневого каталога
    for (thisdir, subshere, fileshere) in os.walk(root): # перечисляет
        print('[' + thisdir + ']')                       # каталоги в дереве
        for fname in fileshere:                      # вывод файлов в каталоге
            path = os.path.join(thisdir, fname)      # добавить имя каталога
            print(path)

if __name__ == '__main__':
    lister(sys.argv[1])                              # имя каталога в
                                                     # командной строке
```

При таком оформлении данный программный код можно также выполнять из командной строки. Ниже приводится пример запуска его для получения списка содержимого другого корневого каталога, который передается в аргументе командной строки:

```
C:\...\PP4E\System\Filetools> python lister_walk.py C:\temp\test
[C:\temp\test]
C:\temp\test\random.bin
```

```
C:\temp\test\spam.txt
C:\temp\test\temp.bin
C:\temp\test\temp.txt
[C:\temp\test\parts]
C:\temp\test\parts\part0001
C:\temp\test\parts\part0002
C:\temp\test\parts\part0003
C:\temp\test\parts\part0004
```

Ниже приводится более сложный пример использования функции `os.walk`. Предположим, что имеется дерево каталогов с файлами, и вам необходимо отыскать в нем все файлы с программным кодом на языке Python, которые ссылаются на модуль `mimetypes` (с этим модулем мы познакомимся в главе 6). Ниже демонстрируется один из способов (хотя и слишком специфичный и не универсальный) решения поставленной задачи:

```
>>> import os
>>> matches = []
>>> for (dirname, dirshere, fileshere) in os.walk(r'C:\temp\PP3E\Examples'):
...     for filename in fileshere:
...         if filename.endswith('.py'):
...             pathname = os.path.join(dirname, filename)
...             if 'mimetypes' in open(pathname).read():
...                 matches.append(pathname)
...
>>> for name in matches: print(name)
...
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py
```

Данная реализация в цикле обходит все файлы в каждом из подкаталогов, отыскивает файлы с расширением *.py*, содержащие искомую строку. Если совпадение найдено, полное имя файла добавляется в объект списка с результатами. Как вариант, мы могли бы просто создать список всех файлов с расширением *.py* и организовать поиск требуемой строки в цикле `for` уже после обхода дерева. Так как в главе 6 мы представим более универсальное решение для этого типа задач, то оставим пока все, как есть.

Если вам будет интересно узнать, что в действительности происходит внутри генератора `os.walk`, попробуйте несколько раз вызвать его метод `__next__` (или передать его встроенной функции `next`), как это автоматически делается циклом `for`, – каждый раз вы будете перемещаться к очередному подкаталогу в дереве:

```
>>> gen = os.walk(r'C:\temp\test')
>>> gen.__next__()
('C:\\temp\\test', ['parts'], ['random.bin', 'spam.txt', 'temp.bin', 'temp.
txt'])
>>> gen.__next__()
('C:\\temp\\test\\parts', [], ['part0001', 'part0002', 'part0003', 'part0004'])
>>> gen.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Описание функции `os.walk` в руководстве по библиотеке содержит более подробную информацию. Например, эта функция поддерживает порядок обхода не только в направлении сверху вниз, но и снизу вверх – достаточно передать функции необязательный аргумент `topdown=False`, и вызывающий программный код получит возможность сократить количество посещаемых ветвей дерева, удаляя имена из списка подкаталогов в возвращаемых кортежах.

Для создания списков имен на каждом уровне в дереве каталогов функция `os.walk` использует функцию `os.listdir`, с которой мы встречались выше, возвращающую имена файлов и каталогов без определенного порядка и без путей к каталогам. Прежде чем вернуть очередной результат, функция `os.walk` делит этот список на списки каталогов и файлов (точнее, некаталогов). Обратите также внимание, что функция `os.walk` использует тот же список подкаталогов, который она возвращает вызывающему программному коду, чтобы затем спуститься в подкаталоги. Списки являются изменяемыми объектами, которые можно изменять непосредственно, поэтому, изменяя содержимое полученного списка подкаталогов, вызывающий программный код может оказывать влияние на дальнейшую работу `os.walk`. Например, удаляя имена каталогов, можно сократить число посещаемых ветвей, а отсортировав список, можно определить очередность обхода подкаталогов.

Рекурсивный обход с помощью os.listdir

Функция `os.walk` сама осуществляет обход дерева – нам остается лишь реализовать тело цикла, выполняющее необходимые действия. Но иногда большей гибкости можно достичь, реализовав обход дерева самостоятельно, при этом почти не приложив лишних усилий. В следующем сценарии представлена другая реализация вывода содержимого каталога с использованием *рекурсивной* функции обхода (функция, которая вызывает саму себя, чтобы повторить операции). Функция `mylister` в примере 4.5 очень похожа на функцию `lister` из примера 4.4, но создает списки имен файлов с помощью `os.listdir` и вызывает саму себя рекурсивно, чтобы спуститься в подкаталоги.

Пример 4.5. PP4E\System\Filetools\lister_recur.py

```
# выводит список файлов в дереве каталогов с применением рекурсии

import sys, os

def mylister(currdir):
    print('[' + currdir + ']')
    for file in os.listdir(currdir):       # генерирует список файлов
        path = os.path.join(currdir, file) # добавить путь к каталогу
        if not os.path.isdir(path):
            print(path)
        else:
            mylister(path)                 # рекурсивный спуск в подкаталоги

if __name__ == '__main__':
    mylister(sys.argv[1])                  # имя каталога в командной строке
```

Как обычно, этот файл можно импортировать или запускать как самостоятельный сценарий. Тот факт, что результатом его работы является печать текста, можно отнести к его недостаткам при его использовании в качестве импортируемого инструмента, если только его стандартный поток вывода не перехватывается в другой программе.

Когда этот файл запускается как самостоятельный сценарий, он воспроизводит почти те же результаты, что и пример 4.4; почти, но не полностью – в отличие от версии на основе функции `os.walk`, рекурсивная версия не обязует пройти все файлы на текущем уровне, прежде чем спуститься в подкаталоги. Можно было бы обойти список имен файлов дважды (чтобы сначала отобрать файлы), но в данной реализации порядок обхода определяется результатами, возвращаемыми функцией `os.listdir`. Для многих случаев такой порядок обхода может оказаться неприемлемым:

```
C:\...\PP4E\System\Filetools> python lister_recur.py C:\temp\test
[C:\temp\test]
[C:\temp\test\parts]
C:\temp\test\parts\part0001
C:\temp\test\parts\part0002
C:\temp\test\parts\part0003
C:\temp\test\parts\part0004
C:\temp\test\random.bin
C:\temp\test\spam.txt
C:\temp\test\temp.bin
C:\temp\test\temp.txt
```

Мы еще воспользуемся большей частью приемов, приведенных в этом разделе, в главе 6 и далее в книге. Например, приведенные выше приемы обхода деревьев будут использованы в сценариях копирования

и сравнения деревьев каталогов. По ходу изложения вы увидите эти инструменты в действии. Кроме того, в главе 6 мы реализуем утилиту *find*, объединяющую в себе обход дерева каталогов с помощью `os.walk` и поиск имен файлов по шаблону с помощью `glob.glob`.

Обработка имен файлов в Юникоде в версии 3.X: listdir, walk, glob

Поскольку в Python 3.X все обычные строки состоят из символов Юникода, имена каталогов и файлов, возвращаемые функциями `os.listdir`, `os.walk` и `glob.glob`, в действительности являются строками Юникода. Это может иметь некоторые последствия, если каталоги содержат необычные имена, не поддающиеся декодированию.

Формально имена файлов могут содержать любые символы, поэтому в версии 3.X функция `os.listdir` может работать в двух режимах: если ей передать аргумент типа `bytes`, она будет возвращать кодированные имена файлов в виде строк байтов; если ей передать аргумент типа `str`, она будет возвращать имена файлов в виде строк Юникода, декодированных в соответствии с кодировкой, используемой файловой системой:

```
C:\...\PP4E\System\Filetools> python
>>> import os
>>> os.listdir('.')[:4]
['bigext-tree.py', 'bigpy-dir.py', 'bigpy-path.py', 'bigpy-tree.py']

>>> os.listdir(b'.')[:4]
[b'bigext-tree.py', b'bigpy-dir.py', b'bigpy-path.py', b'bigpy-tree.py']
```

Версия, основанная на использовании строк байтов, может применяться для файлов с недекодируемыми именами. Функции `os.walk` и `glob.glob` за кулисами обращаются к функции `os.listdir`, от которой наследуют то же самое поведение. Функция `os.walk` обхода деревьев, например, вызывает `os.listdir` для каждого подкаталога – передача строки байтов в аргументе подавляет декодирование, вследствие чего в результате возвращается строка байтов:

```
>>> for (dir, subs, files) in os.walk('..'): print(dir)
...
..
..\Environment
..\Filetools
..\Processes

>>> for (dir, subs, files) in os.walk(b'..'): print(dir)
...
b'..'
b'..\\Environment'
```

```
b'..\\Filetools'
b'..\\Processes'
```

Функция `glob.glob` также вызывает функцию `os.listdir` перед применением шаблонов имен, и поэтому тоже возвращает имена в виде недекодированных строк байтов, когда получает строку байтов в аргументе:

```
>>> glob.glob('.\*')[:3]
['.\\bigext-out.txt', '.\\bigext-tree.py', '.\\bigpy-dir.py']
>>>
>>> glob.glob(b'.\*')[:3]
[b'.\\bigext-out.txt', b'.\\bigext-tree.py', b'.\\bigpy-dir.py']
```

Передавая имена в виде обычных строк (например, посредством аргумента командной строки), вы можете столкнуться с необходимостью преобразовывать обычные строки в строки байтов, с целью подавить декодирование:

```
>>> name = '.'
>>> os.listdir(name.encode())[:4]
[b'bigext-out.txt', b'bigext-tree.py', b'bigpy-dir.py', b'bigpy-path.py']
```

Таким образом, если каталоги могут содержать имена, не поддающиеся декодированию с использованием кодировки, используемой по умолчанию, вам может потребоваться передавать этим инструментам строки байтов, чтобы избежать ошибок, связанных с кодированием Юникода. В ответ вы будете получать строки байтов, которые могут оказаться менее читаемыми при выводе, но это убережет вас от ошибок при обходе каталогов и файлов.

Такой подход может оказаться особенно полезным в системах, где используются простейшие кодировки, такие как ASCII или Latin-1, но могут иметься файлы с именами в произвольных кодировках, скопированными с других компьютеров, из Интернета и так далее. В зависимости от ситуации для подавления некоторых ошибок кодирования можно использовать также обработчики исключений.

Пример того, какое это может иметь значение, мы увидим в первом разделе главы 6, где недекодируемое имя каталога вызывает появление ошибки при выводе во время полного сканирования диска (хотя данная ошибка относится скорее к функции вывода, чем к операции декодирования).

Обратите внимание, что встроенная функция `open` также может принимать имена открываемых файлов как в виде строк `str` Юникода, так и в виде строк байтов `bytes`, однако она использует этот аргумент, только чтобы дать начальное имя файлу, – порядок же обработки содержимого файла определяется дополнительным аргументом режима. Возможность передавать строку байтов в качестве имени файла позволяет использовать произвольные кодированные имена.

Правила использования Юникода: содержимое файлов и имена файлов

Важно помнить, что Юникод может выполнять применительно к файлам две различные задачи: кодирование *содержимого* файлов и кодирование *имен* файлов. Интерпретатор Python определяет настройки по умолчанию для этих двух операций в двух различных атрибутах; для Windows 7:

```
>>> import sys
>>> sys.getdefaultencoding()       # кодировка для содержимого файлов
'utf-8'
>>> sys.getfilesystemencoding()    # кодировка для имен файлов
'mbcs'
```

Эти настройки позволяют явно указывать используемые кодировки – кодировка для содержимого используется операциями чтения из файлов и записи в файлы, а кодировка для имен файлов используется при работе с именами файлов, до передачи данных. Кроме того, использование строк байтов `bytes` для передачи имен файлов различным инструментам позволяет обойти проблему несовместимости со схемой кодирования, используемой файловой системой, а открытие файлов в двоичном режиме позволяет подавить ошибки декодирования их содержимого.

Однако, как мы уже видели выше, открывая текстовые файлы в двоичном режиме, мы можем столкнуться с проблемой несовпадения кодированного текста с искомой строкой в операциях поиска: строки поиска в этом случае также должны быть строками байтов, закодированными с применением определенной кодировки, возможно несовместимой с кодировкой содержимого файла. Фактически данный подход в значительной степени воспроизводит поведение текстовых файлов в Python 2.X и подчеркивает важность использования Юникода в версии 3.X – при работе с такими файлами иногда может сложиться ложное впечатление, что все работает прекрасно. С другой стороны, возможность открывать текстовые файлы в двоичном режиме, чтобы подавить декодирование содержимого файлов и избежать появления связанных с этим ошибок, все еще может быть полезной, если вы не желаете пропустить недекодируемые файлы, содержимое которых не имеет большого значения.

Как правило, необходимо всегда указывать имя кодировки для содержимого текстовых файлов, если она может не совпадать с кодировкой по умолчанию, и в большинстве случаев вам следует опираться на интерфейсы, принимающие имена файлов в виде строк Юникода. Опять же, полную информацию по использованию Юникода в именах файлов вы найдете в руководствах по языку Python, так как здесь недостаточно места, чтобы дать полный охват этой темы, а за информацией

о Юникоде вообще обращайтесь к четвертому изданию книги «Изучаем Python»[1].

В главе 6 мы собираемся задействовать инструменты, с которыми встретились в этой главе, в реальной задаче. Например, мы применим инструменты для работы с файлами и каталогами при реализации делителей файлов, систем тестирования, инструментов копирования и сравнения каталогов, а также других утилит, опирающихся на использование процедуры обхода деревьев. Мы увидим, что инструменты для работы с каталогами, с которыми мы встретились здесь, обладают качествами, позволяющими автоматизировать огромный круг задач. Однако перед этим прочитаем главу 5, завершающую обзор основных инструментов исследованием еще одной темы системного программирования, которая тесно переплетается с различными прикладными областями, – реализацией параллельной обработки данных на языке Python.

[1] Марк Лутц «Изучаем Python», 4 издание, СПб.: Символ-Плюс, 2010.

5

Системные инструменты параллельного выполнения

«Расскажите обезьянам, что им делать»

Большинство компьютеров тратит массу времени, ничего не делая. Если запустить системный монитор и посмотреть на уровень загрузки процессора, вы поймете, что я имею в виду: он очень редко достигает 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 клавиши пользователем и тому подобное. Фактически большая часть мощности современных процессоров большую часть времени не используется: более быстрые процессоры дают ускорение во время пиков потребности в производительности, но значительная часть их мощности в целом может оказаться невостребованной.

Еще на заре эпохи компьютеров программисты поняли, что могут воспользоваться такой неиспользуемой вычислительной мощностью, выполняя одновременно несколько программ. Если распределить процессорное время среди множества задач, его мощность не будет тратиться

---

[1] Для этого в Windows нужно щелкнуть на кнопке Пуск (Start), выбрать пункт меню Все программы (All Programs) → Стандартные (Accessories) → Служебные (System Tools) → Системный монитор (Resource Monitor) и перейти на монитор ЦП (CPU)/Загрузка ЦП (Processor Usage) (аналогичную картину можно наблюдать на вкладке Производительность (Performance) в окне Диспетчера задач (Task Manager)). Когда я писал эту сноску, на моем ноутбуке график не подымался выше 10% (по крайней мере, пока я не ввел `while 1: pass` в окне консоли интерактивного сеанса Python…).

впустую, пока некоторая конкретная задача ждет осуществления внешнего события. Такая технология обычно называется *параллельной обработкой* (или, иногда, «мультиобработкой» или даже «многозадачностью»), потому что возникает впечатление одновременного выполнения нескольких 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 нечто само собой разумеющееся. Даже внутри одной программы 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, выполняемой за кулисами. Обычно недопустимо, чтобы программа зависала при выполнении запроса. Взгляните, например, на пользовательский интерфейс клиента электронной почты: обрабатывая запрос на получение почты с сервера, программа должна загрузить электронные письма 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 останавливаться – она по-прежнему должна реагировать на запросы обновления экрана, щелчки мышью и так далее.

И здесь на помощь приходит параллельная обработка. Выполняя такие долговыполняющиеся задачи параллельно с остальной частью программы, система в целом может сохранить способность реагировать на действия пользователя независимо от того, насколько занятыми оказываются отдельные ее части. Более того, модель параллельной обработки является вполне естественной для структурирования таких и некоторых иных программ – некоторые задачи легче проще проектировать и реализовывать как набор программных компонентов, действующих независимо и параллельно.

Существует два основных способа реализации одновременного выполнения задач в Python – *ветвление процессов* (forks) и *порожденные потоки* (threads) выполнения. Функционально для организации параллельного выполнения программного кода на языке Python оба способа используют службы операционной системы. Процедурно они существенно отличаются в смысле интерфейсов, переносимости и организации взаимодействий между заданиями. Например, на момент написания данной книги возможность прямого ветвления процессов не поддерживалась стандартной реализацией Python для Windows (однако такая поддержка присутствует в версии Python для Cygwin).

Напротив, поддержка потоков выполнения в Python реализована на всех основных платформах. Кроме того, семейство функций `os.spawn`

обеспечивает дополнительные способы запуска способом, не зависящим от типа платформы, – напоминающим ветвление процессов. Для запуска программ переносимым способом, с помощью команд оболочки, также можно использовать функции `os.popen`, `os.system` и модуль `subprocess`, с которыми мы познакомились в главах 2 и 3. Новейший пакет `multiprocessing` предоставляет дополнительные переносимые способы запуска процессов.

В данной главе мы продолжим рассмотрение системных интерфейсов, доступных программистам на языке Python, исследуем встроенные инструменты для параллельного запуска заданий и обмена информацией с этими заданиями. В некотором смысле мы приступили к этому раньше – функции `os.system`, `os. popen` и модуль `subprocess`, которые мы изучали и использовали в предыдущих трех главах, обеспечивают переносимый способ порождения программ командной строки и обмена информацией с ними. Однако здесь мы не собираемся повторять полное описание этих инструментов.

Вместо этого мы сделаем упор на знакомстве с более прямо относящимися к теме приемами, такими как ветвление процессов, потоки, каналы, сигналы, сокеты и другими, и на использовании встроенных инструментов языка Python, поддерживающими их, такими как функция `os.fork` и модули `threading`, `queue` и `multiprocessing`. В следующей главе (и в оставшейся части книги) мы будем использовать эти приемы в примерах действующих программ, поэтому, прежде чем двигаться вперед, необходимо усвоить основы.

Одно предварительное замечание: процессы, потоки и механизмы взаимодействий между процессами, которые мы будем исследовать в этой главе, являются основными инструментами организации параллельной обработки в сценариях на языке Python, однако существует множество сторонних инструментов, предлагающих дополнительные возможности, способные обслуживать расширенные или углубленные потребности. В качестве примера приведу систему MPI для Python, позволяющую в сценариях на языке Python использовать стандартный интерфейс передачи сообщений (Message Passing Interface, MPI), дающий возможность организовать взаимодействие между процессами различными способами (подробности ищите в Интернете). Изучение подобных расширений выходит далеко за рамки этой книги, тем не менее большинство расширенных техник, с которыми вы можете встретиться в будущем, также опираются на основы параллельной обработки, которые мы будем исследовать здесь.

Ветвление процессов

Ветвление процессов является традиционным способом организации параллельных вычислений и представляет собой фундаментальную часть инструментального набора Unix. Ветвление процессов – это са-

мый простой способ запуска независимых программ, как отличных, так и не отличных от вызывающей программы. Прием ветвления основан на понятии *копирования* программ: когда программа вызывает процедуру ветвления, операционная система создает в памяти новую копию этой программы и запускает ее параллельно оригиналу. В некоторых системах исходная программа в действительности не копируется (это слишком дорогостоящая операция), но новая копия работает так, как если бы она действительно была подлинной копией.

После операции ветвления исходный экземпляр программы называется *родительским* процессом, а копия, созданная с помощью функции `os.fork`, называется *дочерним* процессом. Вообще говоря, родитель может воспроизвести любое число потомков, а потомки могут создать собственные дочерние процессы – все ответвленные процессы выполняются независимо и параллельно под управлением операционной системы, и дочерние процессы могут продолжать выполняться даже после завершения родительского процесса.

Возможно, это проще понять на примере, чем в теории. Сценарий Python в примере 5.1 продолжает ответвлять новые дочерние процессы, пока в консоли не будет нажата клавиша `q`.

Пример 5.1. PP4E\System\Processes\fork1.py

```
"ответвляет дочерние процессы, пока не будет нажата клавиша 'q'"

import os

def child():
    print('Hello from child', os.getpid())
    os._exit(0)   # иначе произойдет возврат в родительский цикл

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

parent()
```

Инструменты ветвления процессов в Python, находящиеся в модуле `os`, – это просто тонкие обертки вокруг стандартных средств ветвления из системной библиотеки, используемой также программами на языке C. Запуск нового параллельного процесса осуществляется вызовом функции `os.fork`. Поскольку эта функция создает копию вызывающей программы, она возвращает различные значения в каждой копии: ноль – в дочернем процессе и числовой идентификатор ID процесса нового потомка – в родительском процессе.

Обычно программы проверяют этот результат, чтобы приступить к выполнению каких-то операций только в дочернем процессе. В этом сценарии, например, функция `child` вызывается только в дочерних процессах.[1]

Поскольку ветвление процессов исходно является частью модели программирования в Unix, этот сценарий замечательно будет функционировать в Unix, Linux и в современных версиях Mac OS. К сожалению, этот сценарий не будет работать под управлением стандартной версии Python в Windows, потому что функция `fork` не стыкуется с моделью Windows. Тем не менее в Windows сценарии на языке Python всегда могут порождать потоки выполнения, а также использовать пакет `multiprocessing`, описываемый ниже в этой главе. Этот модуль обеспечивает альтернативный и переносимый способ запуска процессов, который позволяет отказаться от приема ветвления процессов в Windows в контекстах, согласующихся с его ограничениями (хотя и за счет необходимости выполнения некоторых низкоуровневых операций).

Однако сценарий из примера 5.1 будет работать в Windows, если использовать версию Python, распространяемую вместе с системой Cygwin (или собранную вами из исходных текстов вместе с библиотеками Cygwin). Cygwin – это бесплатная и открытая система, обеспечивающая полную Unix-подобную функциональность для Windows (описывается ниже, во врезке «Подробнее о Cygwin Python для Windows»). Используя Python для Cygwin в операционной системе Windows, можно использовать прием ветвления процессов, хотя он не полностью соответствует приему ветвления процессов в Unix. Однако, поскольку эта версия Python достаточно близка к рассматриваемым в данной книге, давайте воспользуемся ею, чтобы запустить сценарий:

```
[C:\...\PP4E\System\Processes]$ python fork1.py
Hello from parent 7296 7920
Hello from child 7920

Hello from parent 7296 3988
Hello from child 3988

Hello from parent 7296 6796
Hello from child 6796
q
```

Эти сообщения представляют три ответвленных дочерних процесса – уникальные идентификаторы всех участвующих процессов получены

[1] По крайней мере, в текущей реализации Python функция `os.fork` в сценарии Python фактически копирует процесс интерпретатора (если взглянуть на список процессов, то после ветвления в нем можно будет найти два процесса Python). Но поскольку интерпретатор Python заботится обо всем, что касается работы сценария, можно считать вызов функции `fork` непосредственным копированием программы. Собственно, так и будет, если скомпилировать сценарии на языке Python в двоичный машинный код.

и выведены с помощью функции os.getpid. Важно отметить, что вызов функции child в дочернем процессе явно завершает его выполнение вызовом функции os._exit. Эту функцию мы более подробно обсудим далее в этой главе, но если ее не вызвать, дочерний процесс продолжит существование после возврата из функции child (не забывайте, что это лишь копия исходного процесса). В этом случае дочерний процесс возвратится в цикл, находящийся в функции parent, и начнет плодить собственных потомков (то есть у родителя появятся внуки). Если удалить вызов выхода и перезапустить сценарий, то для его остановки может понадобиться несколько раз нажать клавишу q, поскольку несколько процессов будут выполнять функцию parent.

В примере 5.1 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 5.2 запускает 5 копий себя самого, при этом каждая копия считает до 5 с односекундной задержкой между итерациями. Функция time.sleep из стандартной библиотеки просто приостанавливает работу вызывающего процесса на указанное количество секунд (допускается указывать значение с плавающей точкой, чтобы приостановить процесс на дробную часть секунды).

Пример 5.2. PP4E\System\Processes\fork-count.py

```
"""
Основы ветвления: запустить 5 копий этой программы параллельно оригиналу; каждая
копия считает до 5 и выводит счетчик в тот же поток stdout -- при ветвлении
копируется память процесса, в том числе дескрипторы файлов; в настоящее время
ветвление не действует в Windows без Cygwin: запускайте программы в Windows
с помощью функции os.spawnv или пакета multiprocessing; функция spawnv примерно
соответствует комбинации функций fork+exec;
"""

import os, time

def counter(count):                         # вызывается в новом процессе
    for i in range(count):
        time.sleep(1)                       # имитировать работу
        print('[%s] => %s' % (os.getpid(), i))

for i in range(5):
    pid = os.fork()
    if pid != 0:                            # в родительском процессе:
        print('Process %d spawned' % pid)   # продолжить цикл
    else:
        counter(5)                          # в дочернем процессе
        os._exit(0)                         # вызвать функцию и завершиться

print('Main process exiting.')              # родитель не должен ждать
```

После запуска этот сценарий сразу запустит 5 процессов и завершит работу. Все 5 ответвленных процессов отображают секундой позже первое показание счетчика и далее – каждую последующую секунду. Обратите внимание, что дочерние процессы продолжают выполняться даже после того, как создавший их родительский процесс завершит свою работу:

```
[C:\...\PP4E\System\Processes]$ python fork-count.py
Process 4556 spawned
Process 3724 spawned
Process 6360 spawned
Process 6476 spawned
Process 6684 spawned
Main process exiting.
[4556] => 0
[3724] => 0
[6360] => 0
[6476] => 0
[6684] => 0
[4556] => 1
[3724] => 1
[6360] => 1
[6476] => 1
[6684] => 1
[4556] => 2
[3724] => 2
[6360] => 2
[6476] => 2
[6684] => 2
```

...остальная часть вывода опущена...

Вывод всех этих процессов отображается на одном и том же экране, потому что все они используют стандартный поток вывода (в процессе работы периодически может появляться системное приглашение к вводу). Технически ответвленный процесс получает копию глобальной памяти оригинального процесса, в том числе дескрипторы открытых файлов. Из-за этого глобальные объекты, такие как файлы, начинают работу в дочернем процессе с одними и теми же значениями, поэтому все процессы в этом примере оказываются подключенными к одному и тому же потоку вывода. Но важно помнить, что глобальная память копируется, а не используется совместно, – если дочерний процесс изменит глобальный объект, то изменит только свою копию этого объекта. (Как мы увидим, в потоках выполнения все происходит совсем иначе. Это тема следующего раздела.)

Комбинация fork/exec

В примерах 5.1 и 5.2 дочерние процессы просто вызывали функцию в программе и завершали свою работу. В Unix-подобных платформах ветвление часто служит основой для запуска программ, выполняю-

щихся независимо и совершенно отличных от программы, вызвавшей функцию fork. Так, в примере 5.3 ответвление новых процессов также выполняется, пока не будет нажата клавиша q, но в дочерних процессах вместо вызова функции в том же файле запускается совершенно новая программа.

Пример 5.3. PP4E\System\Processes\fork-exec.py

```
"запускает программы, пока не будет нажата клавиша 'q'"

import os

parm = 0
while True:
    parm += 1
    pid = os.fork()
    if pid == 0:                                              # копия процесса
        os.execlp('python', 'python', 'child.py', str(parm)) # подменить прогр.
        assert False, 'error starting program'                # возврата быть
                                                              # не должно
    else:
        print('Child is', pid)
        if input() == 'q': break
```

Если вы достаточно много занимались разработкой программ для Unix, комбинация функций fork/exec наверняка будет вам знакома. Главное, на что следует обратить внимание, – это функция os.execlp. В двух словах, эта функция замещает программу, выполняющуюся в текущем процессе, новой программой. Поэтому *комбинация* функций os.fork и os.execlp означает запуск нового процесса и запуск новой программы в этом процессе. Другими словами – запуск новой программы параллельно оригинальной.

Формы вызова функции os.exec

Аргументы функции os.execlp определяют программу, которая должна быть выполнена, и аргументы командной строки, которые следует передать ей (доступные в сценариях Python в виде списка sys.argv). В случае успеха начинается выполнение новой программы, и возврата из вызова функции os.execlp не происходит (так как оригинальная программа замещается новой, то возвращаться действительно некуда). Если возврат все-таки происходит, это означает, что произошла ошибка, поэтому в сценарии после вызова функции стоит инструкция assert, при достижении которой всегда возбуждается исключение.

В стандартной библиотеке Python есть несколько разновидностей функции os.exec. Часть из них позволяет настраивать переменные окружения для новой программы, передавать аргументы командной строки в различных форматах и так далее. Все они имеются как в Unix, так и в Windows, и заменяют вызвавшую их программу (то есть интерпре-

татор Python). Всего существует восемь разновидностей функции exec, что может вызывать затруднения в выборе, если не сделать обобщение:

`os.execv(`*`program, commandlinesequence`*`)`

Базовая «v»-форма функции exec, которой передается имя выполняемой программы вместе со списком или кортежем строк аргументов командной строки, используемых при запуске программы (то есть слов, которые обычно можно ввести в командной строке для запуска программы).

`os.execl(`*`program, cmdargl, cmdarg2, ... cmdargN`*`)`

Базовая «l»-форма функции exec, которой передается имя выполняемой программы, за которым следуют один или более аргументов командной строки, передаваемых как отдельные аргументы функции. Соответствует вызову функции `os.execv(`*`program, (cmdargl, cmdarg2, ...)`*`)`.

`os.execlp`
`os.execvp`

Символ «p», добавленный к именам execv и execl, означает, что Python станет искать каталог, где находится программа, используя системный путь поиска (то есть переменную PATH).

`os.execle`
`os.execve`

Символ «e», добавленный к именам execv и execl, означает, что дополнительный *последний* аргумент является словарем, содержащим переменные окружения, которые нужно передать программе.

`os.execvpe`
`os.execlpe`

Символы «p» и «e», добавленные к базовым именам exec, означают одновременное использование пути поиска *и* словаря с переменными окружения.

Поэтому, когда сценарий в примере 5.3 вызывает os.execlp, отдельно передаваемые параметры определяют аргументы командной строки для программы, которую нужно выполнить, а слово *python* отображается в выполняемый файл, находящийся в пути поиска системы (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 командной оболочке, строка аргументов, передаваемая функции os.execlp сценарием fork-exec из примера 5.3, запускает еще один файл программы Python, который приводится в примере 5.4.

Пример 5.4. PP4E\System\Processes\child.py

```
import os, sys
print('Hello from child', os.getpid(), sys.argv[1])
```

Ниже показано, как этот программный код действует в Linux. Он не сильно отличается от оригинала *fork1.py*, но в действительности запускает *новую программу* в каждом ответвленном процессе. Наиболее наблюдательные читатели заметят, что идентификаторы ID дочернего процесса, отображаемые родительской программой и запущенной программой *child.py*, одинаковые – функция `os.execlp` просто замещает программу в том же самом процессе:

```
[C:\...\PP4E\System\Processes]$ python fork-exec.py
Child is 4556
Hello from child 4556 1

Child is 5920
Hello from child 5920 2

Child is 316
Hello from child 316 3
q
```

В языке Python существуют и другие способы запуска программ, помимо комбинации `fork/exec`. Например, функции `os.system` и `os.popen` и модуль `subprocess`, с которыми мы познакомились в главах 2 и 3, позволяют выполнять команды оболочки. Функция `os.spawnv` и пакет `multiprocessing`, с которым мы познакомимся далее в этой главе, позволяют запускать независимые программы и процессы более переносимым способом. Далее мы увидим, что в некоторых ситуациях модель порождения процессов с помощью пакета `multiprocessing` может использоваться как переносимая замена функции `os.fork` (хотя и менее эффективная) и применяться в соединении с функциями `os.exec*`, показанными здесь, для достижения того же эффекта в стандартной реализации Python для Windows.

Далее в этой главе будут представлены другие примеры ветвления процессов, особенно много – в разделах, посвященных приемам завершения процессов и организации взаимодействий между ними, поэтому мы здесь ограничимся уже приведенными примерами. В следующих главах этой книги мы также рассмотрим другие темы, относящиеся к процессам. Например, в главе 12 мы снова вернемся к приему ветвления процессов, чтобы разобраться с *зомби* – «мертвыми» процессами, затаившимися в системных таблицах после своего конца. А теперь перейдем к потокам выполнения – к теме, которую по крайней мере некоторые программисты находят значительно менее пугающей...

Подробнее о Cygwin Python для Windows

Как уже упоминалось, функция `os.fork` присутствует в версии Cygwin Python для Windows. Эта функция отсутствует в стандартной версии Python для Windows, тем не менее вы можете использовать прием ветвления процессов в Windows, если установите и будете использовать Cygwin. Однако реализация функции `fork` в Cygwin не так эффективна и действует немного не так, как функция `fork` в настоящих системах Unix.

Cygwin – это бесплатный и открытый пакет, включающий библиотеку, реализующую Unix-подобный прикладной интерфейс для использования в Windows, а также набор инструментов командной строки, реализующих Unix-подобное окружение. Это упрощает применение навыков программирования, полученных в Unix, в операционной системе Windows.

Однако, согласно сборнику часто задаваемых вопросов к этому пакету: «Функция `fork()` в Cygwin по сути действует, как некопирующая при записи версия fork() (как это было принято в старых версиях Unix). Вследствие этого она может оказаться немного медленнее. В большинстве случаев лучше использовать семейство функций spawn, когда это возможно». Поскольку производительность не является основной целью примеров в этой книге, будем считать представленную версию функции fork в Cygwin удовлетворительной.

В дополнение к функции fork Cygwin предоставляет и другие инструменты Unix, недоступные ни в одной из версий Windows, включая функцию `os.mkfifo` (обсуждается далее в этой главе). Кроме того, в состав пакета входит компилятор `gcc`, хорошо знакомый разработчикам программ для Unix и позволяющий выполнять сборку расширений на языке C для Python в Windows. Если вы будете использовать библиотеки Cygwin для сборки своих приложений и вашей версии Python, вы окажетесь очень близки к Unix в Windows.

Однако, как и все сторонние библиотеки, Cygwin привносит дополнительную зависимость. Что самое, пожалуй, важное, – Cygwin в настоящее время выходит под лицензией GNU GPL, которая добавляет дополнительные требования к распространению программ, которые гораздо шире требований лицензии для стандартной версии Python. При использовании библиотеки Cygwin в дополнение к самому интерпретатору Python может потребоваться распространять свои программы с открытыми исходными текстами (впрочем, компания RedHat предлагает возможность «выкупа», освобождающую вас от этого требования). Учтите, что это

достаточно сложный юридический вопрос, и вам необходимо внимательно изучить лицензию на Cygwin, которая может распространять свое действие и на ваши программы. Эта лицензия действительно налагает больше ограничений, чем лицензия на Python (Python распространяется под BSD-подобной лицензией, а не GPL).

Но несмотря на проблемы, связанные с лицензией, Cygwin все-таки может служить отличным способом обрести Unix-подобную функциональность в Windows без установки другой полноценной операционной системы, такой как Linux, – более полного, но и более сложного варианта. За дополнительной информацией обращайтесь по адресу *http://cygwin.com* или поищите в Интернете по фразе «Cygwin».

Обратите также внимание на пакет `multiprocessing` из стандартной библиотеки и на семейство функций `os.spawn`, которые будут рассматриваться далее в этой главе. Эти инструменты предоставляют альтернативный способ запуска параллельно выполняющихся заданий и программ в Unix и Windows, которые не требуют наличия в системе функций fork и exec. Чтобы в Windows запустить простую функцию параллельно основной программе (не в виде внешней программы), можно воспользоваться поддержкой потоков выполнения в стандартной библиотеке, о которой рассказывается далее в этой главе. Потоки выполнения, пакет `multiprocessing` и функции `os.spawn` можно использовать в стандартной версии Python для Windows.

Дополнение к четвертому изданию: когда я вносил дополнения в эту главу в феврале 2010 года, в Cygwin официальной версией Python по-прежнему оставалась версия Python 2.5.2. Чтобы получить версию Python 3.1 для Cygwin, ее необходимо собрать из исходных текстов. Если к моменту, когда вы читаете эти строки, данное требование все еще в силе, убедитесь, что в вашем окружении Cygwin установлены компилятор `gcc` и утилита `make`, затем загрузите исходные тексты Python с сайта *python.org*, распакуйте их и соберите Python с помощью следующих команд:

```
./configure
make
make test
sudo make install
```

Эти команды установят Python как `python3`. Ту же процедуру установки можно использовать во всех Unix-подобных системах. В OS X и Cygwin выполняемый файл интерпретатора называется `python.exe`, в остальных окружениях – `python`. Вообще говоря, последние

две команды можно не выполнять, если вы пожелаете запускать Python 3.1 из каталога сборки. Обязательно проверьте, не вошла ли версия Python 3.X в стандартный пакет для Cygwin к тому времени, когда вы будете читать эти строки, – при сборке из исходных текстов вам может потребоваться изменить несколько файлов (мне пришлось закомментировать инструкцию `#define` в файле *Modules/main.c*), однако эти изменения слишком специфические и необходимость в них может отпасть со временем, поэтому я не буду описывать их здесь.

Потоки выполнения

Потоки выполнения представляют еще один способ запуска операций, выполняемых одновременно. В двух словах, механизм потоков выполнения позволяет запустить функцию (или вызываемый объект другого типа) параллельно основной программе. Иногда их называют «облегченными процессами», потому что они работают параллельно, подобно дочерним процессам, но выполняются в рамках одного и того же процесса. Процессы обычно используются для запуска независимых программ, а потоки выполнения – для решения таких задач, как неблокирующий ввод, и для выполнения продолжительных заданий в программах с графическим интерфейсом. Они также представляют естественную модель реализации алгоритмов, которые можно выразить в терминах независимых заданий. В приложениях, которые выигрывают от параллельной обработки, потоки дают программистам большие выгоды:

Производительность

Поскольку все потоки выполняются в пределах одного процесса, их запуск не сопряжен с высокими накладными расходами, как при копировании процесса в целом. Издержки, связанные с копированием порождаемых дочерних процессов и запуском потоков, могут быть различными в зависимости от платформы, но обычно считается, что потоки обходятся дешевле в смысле производительности.

Простота

Потоки выполнения заметно проще в обращении, особенно если на сцену выходят более сложные аспекты процессов (например, завершение процессов, обмен информацией между процессами и процессы-«зомби», о которых рассказывается в главе 12).

Совместно используемая глобальная память

Кроме того, поскольку потоки выполняются в одном процессе, они используют общую глобальную память процесса. Благодаря этому потоки могут просто и естественно взаимодействовать друг с дру-

гом путем чтения и записи данных в глобальной памяти, доступной всем потокам выполнения. Для программиста на языке Python это означает, что глобальные переменные, объекты и их атрибуты и такие компоненты, как импортированные модули, совместно используются всеми потоками выполнения в программе – если, например, в одном потоке выполнения присваивается значение глобальной переменной, ее новое значение увидят все другие потоки выполнения. При обращении к совместно используемым глобальным объектам необходимо проявлять некоторую осторожность, но все равно это обычно проще, чем те средства организации взаимодействий, которые применяются для обмена данными с дочерними процессами и с которыми мы познакомимся ниже в этой главе (например, каналы, потоки ввода-вывода, сигналы, сокеты и так далее). Как и многое в программировании, все вышеизложенное не является универсальной и общепринятой истиной, поэтому вам самим придется взвесить и оценить различия с позиции своих программ и платформ.

Переносимость

Возможно, важнее всего, что приемы работы с потоками выполнения лучше переносятся на другие платформы, чем приемы работы с процессами. На момент написания данной книги функция `os.fork` вообще не поддерживается стандартной версией Python для Windows, тогда как потоки выполнения поддерживаются. Если вам необходимо обеспечить параллельное выполнение заданий в сценариях на языке Python переносимым способом, и вы не желаете или не можете установить в Windows Unix-подобную библиотеку, такую как Cygwin, потоки выполнения окажутся, скорее всего, лучшим решением. Инструменты для работы с потоками выполнения в Python автоматически учитывают специфические для каждой платформы различия в потоках выполнения и предоставляют единообразный интерфейс для всех операционных систем. Следует отметить, что относительно новый пакет `multiprocessing`, описываемый далее в этой главе, предлагает еще одно решение проблемы переносимости, которое может использоваться в некоторых случаях.

Так в чем же подвох? Существует три основных потенциальных недостатка, о которых следует знать, прежде чем нырять в свои потоки выполнения:

Вызовы функций и запуск программ

Прежде всего, потоки выполнения не являются способом, по крайней мере, не самым простым способом, запуска других *программ*. Потоки выполнения предназначены для запуска *функций* (точнее, любого вызываемого объекта, включая связанные и несвязанные методы), выполняющихся параллельно с основной программой. Как мы видели в предыдущем разделе, после выполнения операции ветвления дочерние процессы могут вызывать функции или запускать новые программы. Естественно, функция, запущенная в отдельном

потоке выполнения, также способна запускать другие сценарии с помощью встроенной функции `exec` и новые программы с помощью таких инструментов, как функции `os.system`, `os.popen` и модуль `subprocess`, особенно если они производят продолжительные вычисления. Но вообще, потоки выполнения предназначены для запуска функций внутри программы.

С практической точки зрения это обычно не рассматривается, как недостаток. Для многих приложений возможность параллельного выполнения функций сама по себе является достаточно мощным приобретением. Например, если вам необходимо реализовать неблокирующий ввод и вывод или избежать «подвисания» графического интерфейса из-за выполнения продолжительной операции, с этим прекрасно справятся потоки выполнения – просто создайте поток выполнения. который запустит функцию, производящую продолжительные вычисления, а основная программа продолжит выполняться независимо.

Синхронизация потоков выполнения и очереди

Во-вторых, тот факт, что потоки выполнения совместно используют объекты и переменные в глобальной памяти процесса, имеет свои положительные и отрицательные стороны – это упрощает организацию взаимодействий, но при этом нам необходимо синхронизировать выполнение различных операций. Как мы увидим далее, даже такие операции, как вывод, могут стать источником конфликтов, потому что они пользуются одним потоком вывода `sys.stdout` процесса.

К счастью, модуль `queue` из стандартной библиотеки, описываемый в этом разделе, упрощает решение этой проблемы: на практике многопоточные программы обычно создают один или несколько потоков *производителей* (рабочих потоков), которые добавляют данные в очередь, и один или более потоков *потребителей*, которые извлекают данные из очереди и обрабатывают их. Например, в типичной реализации графического интерфейса производители могут загружать или вычислять данные и помещать их в очередь, а потребитель – главный поток выполнения в программе – периодически проверять наличие данных в очереди по событиям от таймера и отображать их в графическом интерфейсе. Поскольку стандартная реализация очередей уже предусматривает возможность работы с несколькими потоками выполнения, программы, структурированные таким способом, автоматически обеспечивают синхронизацию доступа к данным из нескольких потоков выполнения.

Глобальная блокировка интерпретатора (Global Interpreter Lock, GIL)

Наконец, как мы узнаем далее в этом разделе, реализация механизма потоков выполнения в Python допускает выполнение виртуальной машиной только одного потока в каждый конкретный момент времени. Потоки выполнения в Python являются настоящими пото-

ками выполнения операционной системы, но каждый поток должен приобрести единственную общедоступную блокировку, когда будет готов к запуску, и каждый поток выполнения может быть вытеснен через короткий промежуток времени (в настоящее время – после выполнения виртуальной машиной некоторого количества инструкций, хотя такой порядок может измениться в Python 3.2).

Вследствие этого потоки выполнения в языке Python не могут выполняться одновременно на нескольких процессорах в многопроцессорных системах. Чтобы воспользоваться преимуществами многопроцессорных систем, можно вместо потоков выполнения воспользоваться механизмом ветвления процессов (объем и сложность программного кода в обоих случаях остаются примерно одинаковыми). Кроме того, части потоков выполнения, реализованные как расширения на языке C, могут выполняться по-настоящему независимо, если они освобождают GIL, чтобы обеспечить возможность выполнения программного кода Python в других потоках. Однако программный код на языке Python не может выполняться одновременно в нескольких потоках.

Преимущество реализации механизма потоков выполнения в Python – высокая производительность. Первые попытки внедрить механизм поддержки потоков выполнения в виртуальную машину привели к двукратному снижению скорости выполнения программ в Windows, и еще большее снижение наблюдалось в Linux. Даже однопоточные программы работали в два раза медленнее.

Даже при том, что наличие GIL снижает практическую пользу потоков выполнения в языке Python, не позволяя использовать преимущества многопроцессорных систем, – потоки выполнения остаются полезным инструментом реализации неблокирующих операций, особенно в приложениях с графическим интерфейсом. Кроме того, новый пакет `multiprocessing`, с которым мы познакомимся далее, предлагает другое решение этой проблемы – он предоставляет переносимый прикладной интерфейс, похожий на интерфейс механизма потоков выполнения, но основанный на процессах, благодаря чему программы получают простоту обращения с потоками выполнения и преимущества выполнения независимых процессов в многопроцессорных системах.

Несмотря на то, что после прочтения этого обзора у вас могло сложиться иное мнение, я утверждаю, что потоки выполнения в языке Python удивительно просты в использовании. Фактически когда запускается программа, она уже выполняется в потоке, который обычно называется «главным потоком» процесса. Для запуска новых, независимых потоков выполнения в рамках одного и того же процесса в программах на языке Python обычно используется либо низкоуровневый модуль `_thread`, позволяющий запускать функции в порожденных потоках выполнения, либо высокоуровневый модуль `threading`, предоставляющий возмож-

ность управления потоками выполнения с помощью объектов высокого уровня, созданных на основе классов. Оба модуля также предусматривают инструменты синхронизации доступа к совместно используемым объектам с помощью блокировок.

В данной книге будут исследоваться оба модуля, `_thread` и `threading`, и в примерах они будут использоваться взаимозаменяемо. Некоторые программисты на языке Python могли бы порекомендовать всегда использовать модуль `threading` и оставить модуль `_thread` в покое. Последний из них ранее назывался `thread` и в версии 3.X получил название `_thread`, которое предполагает менее высокий статус модуля. Лично я считаю, что это крайность (это одна из причин, почему в некоторых примерах в данной книге используется конструкция `as thread` в инструкциях импортирования, позволяющая использовать оригинальное имя модуля в программном коде). Если только вам не требуются мощные инструменты из модуля `threading`, выбор между этими двумя модулями является вопросом личных предпочтений, при этом дополнительные требования модуля `threading` могут считаться ничем не оправданными.

В базовом модуле `_thread` не используются приемы объектно-ориентированного программирования, и он очень прост в использовании, как будет показано в примерах этого раздела. Модуль `threading` лучше подходит для решения более сложных задач, которые требуют сохранения информации в контексте потоков или наблюдения за потоками, но не все многопоточные программы требуют применения дополнительных инструментов, и во многих из них используется достаточно ограниченный набор возможностей многопоточной модели. Фактически сравнение этих модулей напоминает сравнение функции `os.walk` с классами, реализующими обход дерева, с которыми мы встретимся в главе 6, – оба приема имеют своих сторонников и область применения. Как всегда, не забывайте основное правило Python: *не добавляйте сложностей, когда сложности не нужны*.

Модуль _thread

Поскольку базовый модуль `_thread` немного проще, чем более мощный модуль `threading`, о котором рассказывается далее в этом разделе, начнем с рассмотрения его интерфейсов. Этот модуль предоставляет *переносимый* интерфейс к любой системе потоков выполнения, имеющейся на вашей платформе: его интерфейсы одинаково работают в Windows, Solaris, SGI и любой другой системе, где установлена реализация `pthreads` потоков POSIX (включая Linux). Сценарии на языке Python, использующие модуль `_thread`, будут работать на всех этих платформах без внесения каких-либо изменений в исходный программный код.

Основы использования

Для начала поэкспериментируем со сценарием, демонстрирующим применение основных интерфейсов механизма потоков выполнения. Сценарий в примере 5.5 порождает потоки выполнения, пока в консоли не будет нажата клавиша q, и напоминает по духу (будучи немного проще) сценарий в примере 5.1; но он запускает параллельно потоки, а не дочерние процессы.

Пример 5.5. PP4E\System\Threads\thread1.py

```
"порождает потоки выполнения, пока не будет нажата клавиша 'q'"

import _thread

def child(tid):
    print('Hello from thread', tid)

def parent():
    i = 0
    while True:
        i += 1
        _thread.start_new_thread(child, (i,))
        if input() == 'q': break

parent()
```

В действительности в этом сценарии только две строки имеют отношение к потокам выполнения: инструкция импортирования модуля `_thread` и вызов функции, создающей поток. Чтобы запустить новый поток выполнения, достаточно просто вызвать функцию `_thread.start_new_thread`, независимо от того, на какой платформе выполняется программа.[1] Эта функция принимает функцию (или другой вызываемый объект) и кортеж аргументов, и запускает новый поток выполнения, в котором будет вызвана указанная функция с переданными аргументами. Это очень похоже на синтаксис вызова `function(*args)` – и тут, и там принимается необязательный словарь именованных аргументов, – но в данном случае функция начинает выполняться параллельно основной программе.

Сама функция `_thread.start_new_thread` сразу же возвращает управление вызывающей, не возвращая какого-либо полезного значения, а порож-

[1] В примерах использования модуля `_thread` в этой книге теперь везде используется функция `start_new_thread`. По исторически сложившимся причинам эта функция доступна также под именем `thread.start_new`, но в будущих версиях Python этот синоним может быть удален. В версии Python 3.1 оба имени по-прежнему доступны, но в документации, которая выводится функцией `help`, функция `start_new` объявлена устаревшей. Другими словами, ее не следует использовать, если вы беспокоитесь о будущем (и что должно учитываться в книге!).

денный ею поток тихо завершается, когда происходит возврат из выполняемой функции (значение, возвращаемое функцией, выполняемой в потоке, просто игнорируется). Кроме того, если выполняемая в потоке функция возбудит исключение, интерпретатор выведет трассировочную информацию и завершит работу потока, но остальная программа продолжит работу. На большинстве платформ при использовании модуля `_thread` вся программа завершит работу без вывода каких-либо сообщений, когда завершится главный поток (однако, как будет показано далее, при использовании модуля `threading` может потребоваться предпринять дополнительные действия, если дочерние потоки к этому моменту еще продолжают выполняться).

На практике, однако, использование потоков выполнения в сценариях на языке Python почти тривиально. Запустим эту программу и позволим ей породить несколько новых потоков. На этот раз ее можно выполнять как в Unix-подобных системах, так и в Windows, потому что потоки переносятся лучше, чем ветвление процессов. Ниже приводится пример порождения потоков в Windows:

```
C:\...\PP4E\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 выполнения, который завершается почти сразу после запуска.

Другие способы реализации потоков с помощью модуля _thread

В предыдущем примере сценарий запускает простую функцию, тем не менее в отдельном потоке выполнения можно запустить *любой вызываемый объект*, благодаря тому что все потоки выполняются в рамках одного и того же процесса. Например, в отдельном потоке можно запустить `lambda`-функцию или связанный метод объекта (ниже приводится фрагмент сценария *thread-alts.py*, входящего в состав пакета с примерами к книге):

```
import _thread                                  # во всех 3 случаях
                                                # выводится 4294967296

def action(i):                                  # простая функция
    print(i ** 32)

class Power:
    def __init__(self, i):
```

```
        self.i = i
    def action(self):                              # связанный метод
        print(self.i ** 32)

_thread.start_new_thread(action, (2,))             # запуск простой функции

_thread.start_new_thread((lambda: action(2)), ())  # запуск lambda-функции

obj = Power(2)
_thread.start_new_thread(obj.action, ())           # запуск связанного метода
```

Как будет показано далее в книге, в более крупных примерах, в этой роли особенно полезными оказываются *связанные методы* – так как они хранят в себе и ссылку на функцию, и ссылку на экземпляр объекта, то они обладают доступом к информации о состоянии и методам класса, которые могут использовать в процессе выполнения внутри потока.

Если смотреть глубже – так как все потоки выполняются в рамках одного и того же процесса, то связанные методы, выполняемые в отдельных потоках, имеют доступ к оригинальному экземпляру объекта, а не к его копии. Следовательно, любые изменения, выполненные в потоке, автоматически будут видимы для всех остальных потоков. Кроме того, связанные методы экземпляров классов, как вызываемые объекты, могут использоваться вместо простых функций, поэтому использование их в потоках выполнения не влечет никаких сложностей. И, как будет показано далее, тот факт, что они являются обычными объектами, позволяет сохранять их в общедоступных очередях.

Запуск нескольких потоков

По-настоящему ощутить всю мощь параллельно выполняющихся потоков можно, только если реализовать в них выполнение продолжительных операций, как мы делали это выше для процессов. Изменим программу `fork-count` из предыдущего раздела так, чтобы в ней использовались потоки выполнения. В сценарии из примера 5.6 запускается 5 экземпляров функции `counter`, которые выполняются параллельно в отдельных потоках.

Пример 5.6. PP4E\System\Threads\thread-count.py

```
"""
основы потоков: запускает 5 копий функции в параллельных потоках; функция time.
sleep используется, чтобы главный поток не завершился слишком рано, так как
на некоторых платформах это приведет к завершению остальных потоков выполнения;
поток вывода stdout - общий: результаты, выводимые потоками выполнения, в этой
версии могут перемешиваться произвольным образом.
"""

import _thread as thread, time
```

```
def counter(myId, count):                    # эта функция выполняется в потоках
    for i in range(count):
        time.sleep(1)                        # имитировать работу
        print('[%s] => %s' % (myId, i))

for i in range(5):                           # породить 5 потоков выполнения
    thread.start_new_thread(counter, (i, 5)) # каждый поток выполняет 5 циклов

time.sleep(6)
print('Main thread exiting.')                # задержать выход из программы
```

Каждая параллельно выполняющаяся копия функции `counter` просто считает здесь от нуля до четырех и при каждом увеличении счетчика выводит сообщение в поток стандартного вывода.

Обратите внимание, что в самом конце этот сценарий приостанавливается на 6 секунд. В Windows и в Linux, как было проверено, главный поток не должен завершаться, пока все порожденные потоки не закончили работу, если важно, чтобы они доработали. Если главный поток завершится раньше, все порожденные потоки будут немедленно завершены. Этим потоки выполнения отличаются от процессов, где дочерние процессы продолжают работать после завершения родительского процесса. Если убрать вызов функции `sleep` в конце сценария, порожденные потоки выполнения будут немедленно завершены, практически сразу же после их запуска.

Может показаться, что так сделано специально, но это необходимо не на всех платформах, и программы обычно реализованы так, чтобы главный поток выполнения продолжал работать столько же, сколько потоки, им запущенные. Например, интерфейс пользователя может начать загрузку файла по протоколу FTP в потоке, но продолжительность операции загрузки значительно короче, чем время жизни самого интерфейса пользователя. Далее в этом разделе мы увидим, как различными способами можно избежать этой паузы с помощью глобальных блокировок и флагов, позволяющих потокам выполнения сигнализировать о своем завершении.

Кроме того, далее мы узнаем, что модуль `threading` предоставляет метод `join`, который позволяет дождаться завершения порожденных потоков и не дает программе завершиться до того, пока хотя бы один обычный поток выполнения продолжает работу (что было бы полезно в данном случае, но в других случаях может потребовать выполнения дополнительных операций по принудительному завершению потоков). Пакет `multiprocessing`, с которым мы встретимся далее в этой главе, также позволяет потомкам продолжать работу после завершения родителя, но это в значительной степени объясняется использованием модели процессов.

Если теперь запустить сценарий из примера 5.6 в Windows 7 под управлением Python 3.1, он выведет:

```
C:\...\PP4E\System\Threads> python thread-count.py
[1] => 0
[1] => 0
[0] => 0
[1] => 0
[0] => 0
[2] => 0
[3] => 0
[3] => 0

[1] => 1
[3] => 1
[3] => 1
[0] => 1[2] => 1
[3] => 1
[0] => 1[2] => 1
[4] => 1

[1] => 2
[3] => 2[4] => 2
[3] => 2[4] => 2
[0] => 2
[3] => 2[4] => 2
[0] => 2
[2] => 2
[3] => 2[4] => 2
[0] => 2
[2] => 2
...часть вывода опущена...
Main thread exiting.
```

Полученные результаты, возможно, покажутся вам странными, но так они и должны выглядеть. Данный пример демонстрирует один из наиболее необычных аспектов потоков выполнения. В этом примере результаты 5 потоков, действующих параллельно, перемешались между собой. Поскольку все потоки выполняются в рамках одного и того же процесса, все они совместно используют один и тот же поток стандартного вывода (в терминах языка Python они совместно используют файл `sys.stdout`, куда выводит текст функция `print`). В результате вывод потоков выполнения может перемешиваться произвольно. На практике при каждом запуске этого сценария могут быть получены разные результаты. В Python 3 перемешивание вывода стало еще более явным, что, вероятно, обусловлено новой реализацией вывода в файлы.

Из этого следует важный вывод: когда несколько потоков выполнения могут совместно использовать некоторый ресурс, как в данном примере, операции доступа в них должны синхронизироваться, чтобы избежать перекрытия во времени, а как – будет описано в следующем разделе.

Синхронизация доступа к глобальным объектам и переменным

Приятной особенностью потоков выполнения является наличие готового механизма обмена данными между заданиями: объектов и переменных процесса, совместно используемых потоками. Например, поскольку все потоки выполняются в одном и том же процессе, то при изменении глобальной переменной одним потоком это изменение видно всем другим потокам в процессе: и главному потоку, и дочерним. Точно так же потоки могут совместно использовать изменяемые объекты в памяти процесса, при условии, что они хранят ссылки на них (например, полученные в виде аргументов). Это упрощает возможность передачи информации между потоками программы – флаги завершения, объекты с результатами, индикаторы событий и так далее.

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 будет утрачено (или, что еще хуже, совместно используемый объект придет в полностью негодное состояние): один поток может приступить к выполнению операций, когда другой поток еще не завершил работу с объектом. Приводит ли это к ошибке, зависит от приложения; иногда проблем вообще не возникает.

Проблемы могут возникнуть и там, где этого совсем не ждешь. Например, файлы и потоки ввода-вывода совместно используются всеми потоками выполнения программы – если несколько потоков выполнения одновременно производят запись в один и тот же поток ввода-вывода, в последнем могут появиться перемежающиеся искаженные данные. Пример 5.6 из предыдущего раздела является простой, но показательной демонстрацией подобного рода конфликтов, которые могут происходить при параллельном выполнении нескольких потоков. Даже простейшие изменения могут пустить все вкривь и вкось, когда есть вероятность одновременного их выполнения. Чтобы исключить подобные ошибки, программы должны управлять доступом к глобальным объектам, чтобы в каждый конкретный момент времени только один поток выполнения мог использовать их.

К счастью, в модуле `_thread` имеются собственные простые в использовании инструменты синхронизации потоков, выполняющих операции с совместно используемыми объектами. Эти инструменты основаны на понятии *блокировки* – чтобы изменить совместно используемый объект, потоки *приобретают* блокировку, производят требуемые изменения и *освобождают* блокировку для использования в других потоках выполнения. Интерпретатор гарантирует, что в каждый конкретный момент времени только один поток выполнения будет владеть блокировкой, – если запрос на приобретение блокировки поступит в тот момент, когда она удерживается некоторым потоком, запросивший поток будет приостановлен до того момента, пока блокировка не будет освобождена.

Объекты блокировки размещаются в памяти, обрабатываются с помощью простых и переносимых функций из модуля _thread и автоматически отображаются на механизмы блокировки потоков, существующие на соответствующей платформе.

Так, в примере 5.7 с помощью функции _thread.allocate_lock создается объект блокировки, который приобретается и освобождается каждым потоком выполнения перед вызовом функции print, с помощью которой осуществляется вывод в совместно используемый стандартный поток вывода.

Пример 5.7. PP4E\System\Threads\thread-count-mutex.py

```
"""
синхронизирует доступ к stdout: так как это общий глобальный объект, данные,
которые выводятся из потоков выполнения, могут перемешиваться, если  не
синхронизировать операции
"""

import _thread as thread, time

def counter(myId, count):                  # эта функция выполняется в потоках
    for i in range(count):
        time.sleep(1)                        # имитировать работу
        mutex.acquire()
        print('[%s] => %s' % (myId, i))      # теперь работа функции print
                                             # не будет прерываться
        mutex.release()

mutex = thread.allocate_lock()               # создать объект блокировки
for i in range(5):                           # породить 5 потоков выполнения
    thread.start_new_thread(counter, (i, 5)) # каждый поток выполняет 5 циклов

time.sleep(6)
print('Main thread exiting.')                # задержать выход из программы
```

В действительности этот сценарий является всего лишь расширенной версией примера 5.6, в которую была добавлена синхронизация обращений к функции print с применением блокировки. Благодаря этому никакие два потока выполнения в этом сценарии не смогут одновременно вызвать функцию print – блокировка гарантирует исключительный доступ к стандартному потоку вывода stdout. Таким образом, мы получаем вывод, сходный с выводом оригинальной версии, за исключением того, что текст на выходе никогда не будет перемешиваться из-за перекрывающихся операций вывода:

```
C:\...\PP4E\System\Threads> thread-count-mutex.py
[0] => 0
[1] => 0
[3] => 0
[2] => 0
[4] => 0
```

```
[0] => 1
[1] => 1
[3] => 1
[2] => 1
[4] => 1
[0] => 2
[1] => 2
[3] => 2
[4] => 2
[2] => 2
[0] => 3
[1] => 3
[3] => 3
[4] => 3
[2] => 3
[0] => 4
[1] => 4
[3] => 4
[4] => 4
[2] => 4
Main thread exiting.
```

Порядок, в каком потоки выполнения выводят свои данные, зависит от платформы и по-прежнему может изменяться от запуска к запуску, потому что они выполняются параллельно (в конце концов, потоки выполнения как раз и предназначены для параллельной обработки данных). Но они больше не конфликтуют при выводе текста. Далее в этой главе мы увидим другие случаи использования блокировок – блокировки являются важной составляющей многопоточной модели выполнения.

Ожидание завершения порожденных потоков выполнения

Помимо устранения конфликтов при выводе данных, блокировки модуля потоков имеют и другие очень полезные применения. Они могут использоваться в качестве основы парадигм синхронизации более высокого уровня (например, семафоров) и использоваться как универсальные инструменты взаимодействий между потоками.[1] В частности,

[1] Однако их нельзя использовать для непосредственной синхронизации *процессов*. Поскольку процессы более независимы, им обычно требуются механизмы блокировки, более долговременные и внешние по отношению к программам. Вызов функции `os.open`, представленной в главе 4, с флагом `O_EXCL` позволяет сценариям блокировать и разблокировать доступ к файлам и потому может считаться идеальным инструментом блокировки для использования в процессах. Обратите также внимание на инструменты синхронизации в модулях `multiprocessing` и `threading`, а также прочитайте раздел об организации взаимодействий между процессами далее в этой главе, где представлены другие универсальные приемы синхронизации.

в примере 5.8 глобальный список блокировок позволяет установить окончание работы всех дочерних потоков.

Пример 5.8. PP4E\System\Threads\thread-count-wait1.py

```
"""
использование мьютексов в родительском/главном потоке выполнения для определения
момента завершения дочерних потоков, взамен time.sleep; блокирует stdout, чтобы
избежать конфликтов при выводе;
"""

import _thread as thread
stdoutmutex = thread.allocate_lock()
exitmutexes = [thread.allocate_lock() for i in range(10)]

def counter(myId, count):
    for i in range(count):
        stdoutmutex.acquire()
        print('[%s] => %s' % (myId, i))
        stdoutmutex.release()
    exitmutexes[myId].acquire()                # сигнал главному потоку

for i in range(10):
    thread.start_new_thread(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`. Главный поток создает по одной блокировке для каждого дочернего потока, помещая их в глобальный список `exitmutexes` (не забывайте, что функция потока использует глобальную область совместно с главным потоком). По завершении каждый поток приобретает свою блокировку в списке, а главный поток просто ждет, когда будут приобретены все блокировки. Это значительно более точный подход, чем просто приостанавливать работу на определенное время, пока выполняются дочерние потоки, в надежде обнаружить после возобновления, что все они будут завершены.

В зависимости от операций, выполняемых в потоках, все это можно организовать 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 5.9 пространство имен модуля (область видимости), как и прежде, совместно используется программным кодом верхнего уровня и функцией, выполняемой в потоке. Имя `exitmutexes` ссылается на один и тот же объект списка в главном потоке и во всех порождаемых потоках. По этой причине изменения, производимые в потоке, видны в главном потоке без использования лишних блокировок.

Пример 5.9. PP4E\System\Threads\thread-count-wait2.py

```
"""
использование простых глобальных данных (не мьютексов) для определения момента
завершения всех потоков в родительском/главном потоке; потоки совместно
используют список, но не его элементы, при этом предполагается, что после
создания список не будет перемещаться в памяти
"""

import _thread as thread
stdoutmutex = thread.allocate_lock()
exitmutexes = [False] * 10

def counter(myId, count):
    for i in range(count):
        stdoutmutex.acquire()
        print('[%s] => %s' % (myId, i))
        stdoutmutex.release()
    exitmutexes[myId] = True                    # сигнал главному потоку

for i in range(10):
    thread.start_new_thread(counter, (i, 100))

while False in exitmutexes: pass
print('Main thread exiting.')
```

Вывод этого сценария похож на вывод предыдущего – 10 потоков параллельно ведут счет до 100 и в процессе работы синхронизируют свои обращения к функции `print`. Фактически оба последних сценария с потоками-счетчиками производят вывод, в общем аналогичный первоначальному сценарию *thread_count.py*, но данные при выводе в `stdout` не повреждаются, значения счетчиков больше и отличается случайный порядок вывода строк. Основное отличие состоит в том, что главный поток завершает работу сразу после (и не раньше!) порожденных дочерних потоков:

```
C:\...\PP4E\System\Threads> python thread-count-wait2.py
...часть вывода удалена...
[4] => 98
[6] => 98
[8] => 98
[5] => 98
[0] => 99
[7] => 98
[9] => 98
[1] => 99
[3] => 99
[2] => 99
[4] => 99
[6] => 99
[8] => 99
[5] => 99
```

```
[7] => 99
[9] => 99
Main thread exiting.
```

Альтернативные приемы: циклы занятости, аргументы и менеджеры контекста

Обратите внимание, что главные потоки выполнения в двух последних сценариях в конце выполняют цикл ожидания, который может заметно снизить производительность в критически важных приложениях. В таких ситуациях достаточно просто добавить в цикл ожидания вызов функции `time.sleep`, чтобы оформить паузу между проверками и освободить процессор для других заданий: эта функция будет приостанавливать только вызывающий поток выполнения (в данном случае – главный поток). Можно также попробовать добавить вызов функции `sleep` в функцию, которая выполняется в потоках, чтобы сымитировать выполнение продолжительных операций.

Для единообразия вместо использования глобальной области видимости можно было бы также организовать передачу блокировки в виде *аргумента* функции, которая выполняется в потоках. В этом случае все потоки выполнения будут ссылаться на один и тот же объект блокировки, потому что все они являются частью одного и того же процесса. Память процесса, занятая объектом, является памятью, совместно используемой потоками независимо от того, как будет получена ссылка на этот объект (через глобальные переменные, через аргументы функции, через атрибуты объектов или каким-либо другим способом).

И еще – чтобы гарантировать освобождение блокировки при выходе потока выполнения из критического блока, можно использовать инструкцию `with`, как мы делали это в предыдущей главе, чтобы обеспечить закрытие файлов. *Менеджер контекста* блокировки приобретает блокировку при входе в инструкцию `with` и освобождает ее при выходе из тела инструкции, независимо от того, возникло исключение или нет. Этот прием позволяет сэкономить одну строку программного кода и дополнительно гарантирует освобождение блокировки в ситуациях, когда возможно появление исключения. Все эти приемы реализованы в примере 5.10, представляющем улучшенную версию нашего сценария с потоками-счетчиками.

Пример 5.10. PP4E\System\Threads\thread-count-wait3.py

```
"""
объект мьютекса, совместно используемый всеми потоками выполнения, передается
функции в виде аргумента; для автоматического приобретения/освобождения
блокировки используется менеджер контекста; чтобы избежать излишней нагрузки
в цикле ожидания, и для имитации выполнения продолжительных операций добавлен
вызов функции sleep
"""

import _thread as thread, time
```

```
stdoutmutex = thread.allocate_lock()
numthreads  = 5
exitmutexes = [thread.allocate_lock() for i in range(numthreads)]

def counter(myId, count, mutex):   # мьютекс передается в аргументе
    for i in range(count):
        time.sleep(1 / (myId+1))   # различные доли секунды
        with mutex:                # приобретает/освобождает блокировку: with
            print('[%s] => %s' % (myId, i))
    exitmutexes[myId].acquire()    # глобальный список: сигнал главному потоку

for i in range(numthreads):
    thread.start_new_thread(counter, (i, 5, stdoutmutex))

while not all(mutex.locked() for mutex in exitmutexes): time.sleep(0.25)
print('Main thread exiting.')
```

Различные времена ожидания для разных потоков выполнения делают их более независимыми:

```
C:\...\PP4E\System\Threads> thread-count-wait3.py
[4] => 0
[3] => 0
[2] => 0
[4] => 1
[1] => 0
[3] => 1
[4] => 2
[2] => 1
[3] => 2
[4] => 3
[4] => 4
[0] => 0
[1] => 1
[2] => 2
[3] => 3
[3] => 4
[2] => 3
[1] => 2
[2] => 4
[0] => 1
[1] => 3
[1] => 4
[0] => 2
[0] => 3
[0] => 4
Main thread exiting.
```

Конечно, потоки выполнения могут решать гораздо более сложные задачи, чем простой подсчет. Более практичный пример использования глобальных данных мы рассмотрим в разделе «Добавляем пользовательский интерфейс» в главе 13, где они будут играть роль сигналов

главному потоку, управляющему графическим интерфейсом на основе библиотеки tkinter, о завершении дочерним потоком передачи данных по сети, а также в главе 10, в примере реализации модуля threadtools, и в главе 14, в примере приложения PyMailGUI, для отображения результатов отправки электронной почты в графическом интерфейсе (дополнительные указания по этой теме вы найдете в разделе «Графические интерфейсы и потоки выполнения: предварительное знакомство» ниже, в этой главе). Возможность совместного доступа к глобальным данным из потоков выполнения также является основой организации очередей, которые обсуждаются далее в главе, – каждый поток выполнения может извлекать или добавлять данные, используя один и тот же общий объект очереди.

Модуль threading

В стандартную библиотеку Python входят два модуля для работы с потоками: _thread – основной низкоуровневый интерфейс, который демонстрировался до сих пор, и threading – интерфейс более высокого уровня, основанный на объектах и классах. Внутри модуль threading использует модуль _thread для реализации объектов, представляющих потоки и инструменты синхронизации. Он в какой-то мере основан на подмножестве модели потоков выполнения языка Java, но есть различия, которые заметят только программисты Java.[1] В примере 5.11 приводится еще одна, последняя версия нашего сценария с потоками-счетчиками, демонстрирующая интерфейсы этого нового модуля.

Пример 5.11. PP4E\System\Threads\thread-classes.py

```
"""
экземпляры класса Thread, сохраняющие информацию о состоянии и обладающие
методом run() для запуска потоков выполнения; в реализации используется
высокоуровневый и Java-подобный метод join класса Thread модуля threading
(вместо мьютексов и глобальных переменных), чтобы известить главный родительский
поток о завершении дочерних потоков; подробности о модуле threading ищите
в руководстве по стандартной библиотеке;
"""

import threading

class Mythread(threading.Thread):        # подкласс класса Thread
    def __init__(self, myId, count, mutex):
        self.myId = myId
        self.count = count               # информация для каждого потока
        self.mutex = mutex               # совместно используемые объекты,
```

[1] Пояснение для программистов Java: блокировки и условные переменные в языке Python являются отдельными объектами, а не чем-то присущим всем объектам, а класс Thread в языке Python не обладает всеми характеристиками, которые есть у одноименного класса в языке Java. Дополнительные детали можно найти в руководстве по библиотеке Python.

```
        threading.Thread.__init__(self)    # вместо глобальных переменных
    def run(self):                         # run реализует логику потока
        for i in range(self.count):        # синхронизировать доступ к stdout
            with self.mutex:
                print('[%s] => %s' % (self.myId, i))

stdoutmutex = threading.Lock()             # то же, что и thread.allocate_lock()
threads = []
for i in range(10):
    thread = Mythread(i, 100, stdoutmutex) # создать/запустить 10 потоков
    thread.start()                         # вызвать метод run потока
    threads.append(thread)

for thread in threads:
    thread.join()                          # ждать завершения потока
print('Main thread exiting.')
```

Этот сценарий производит точно такой же вывод, как и его предшественники (и снова строки случайно распределены по времени, в зависимости от используемой платформы):

```
C:\...\PP4E\System\Threads> python thread-classes.py
...часть вывода удалена...
[4] => 98
[8] => 97
[9] => 97
[5] => 98
[3] => 99
[6] => 98
[7] => 98
[4] => 99
[8] => 98
[9] => 98
[5] => 99
[6] => 99
[7] => 99
[8] => 99
[9] => 99
Main thread exiting.
```

Использование модуля `threading` заключается в основном в определении новых классов. Потоки в этом модуле реализуются с помощью объекта `Thread` – класса Python, который наследуется и специализируется в каждом приложении путем реализации метода `run`, определяющего действия, выполняемые потоком. Например, в данном сценарии создается подкласс `Mythread` класса `Thread`, метод `run` которого будет вызываться родительским классом `Thread` в новом потоке после создания экземпляра класса `Mythread` и вызова его метода `start`.

Иными словами, этот сценарий просто обеспечивает методы, предполагаемые структурой класса `Thread`. Преимущество этого приема, требующего создания большего объема программного кода, заключается в том,

что он обеспечивает «бесплатный» доступ к информации о состоянии каждого потока в отдельности (в виде атрибутов экземпляра) и к ряду дополнительных инструментов для работы с потоками, предоставляемых данной структурой. К примеру, используемый в конце сценария метод Thread.join ожидает завершения (по умолчанию) потока выполнения – этот метод можно использовать, чтобы предотвратить завершение главного потока до того, как завершится дочерний поток, и отказаться от вызова функции time.sleep, глобальных блокировок и переменных, использовавшихся в предыдущих примерах с потоками.

Кроме того, для синхронизации доступа к стандартному потоку вывода в примере 5.11 используется конструктор threading.Lock (хотя в текущей реализации это просто синоним конструктора _thread.allocate_lock). Модуль threading предоставляет и другие структуры классов, но они не влияют на общую картину многопоточной модели параллельной обработки данных.

Другие способы реализации потоков выполнения с помощью модуля threading

Класс Thread можно также использовать для запуска простых функций и вызываемых объектов других типов, вообще не создавая подклассы. Метод run класса Thread по умолчанию просто вызывает объект, переданный конструктору в аргументе target, со всеми дополнительными аргументами, переданными в аргументе args (который по умолчанию является пустым списком ()). Это позволяет использовать класс Thread для запуска простых функций, хотя такая форма вызова ненамного проще использования модуля _thread. Например, в следующих фрагментах демонстрируются четыре различных способа запуска одного и того же потока (смотрите сценарии *four-threads*.py* в дереве примеров; вы можете запустить все четыре потока в одном сценарии, но при этом вам понадобится синхронизировать обращения к функции print, чтобы избежать смешивания выводимых данных):

```
import threading, _thread
def action(i):
    print(i ** 32)

# подкласс, хранящий собственную информацию о состоянии
class Mythread(threading.Thread):
    def __init__(self, i):
        self.i = i
        threading.Thread.__init__(self)
    def run(self):                          # переопределить метод run
        print(self.i ** 32)
Mythread(2).start()                         # метод start вызовет метод run()

# передача простой функции
thread = threading.Thread(target=(lambda: action(2)))     # run вызовет target
thread.start()
```

```
# то же самое, но без lambda-функции,
# сохраняющей информацию о состоянии в образуемом ею замыкании
threading.Thread(target=action, args=(2,)).start()      # вызываемый объект
                                                        # и его аргументы
# с помощью модуля thread
_thread.start_new_thread(action, (2,))        # полностью процедурный интерфейс
```

Как правило, выбирать реализацию потоков на основе классов имеет смысл, когда потоки должны сохранять информацию о своем состоянии или когда желательно использовать какие-либо из многочисленных преимуществ ООП. Однако классы потоков выполнения необязательно должны наследовать класс `Thread`. Фактически, как и при использовании модуля `_thread`, реализация потоков в модуле `threading` может принимать в аргументе `target` *вызываемые объекты любого типа*. При объединении с такими приемами, как связанные методы и вложенные области видимости, различия между приемами программирования становятся еще менее выраженными:

```
# обычный класс с атрибутами, ООП
class Power:
    def __init__(self, i):
        self.i = i
    def action(self):
        print(self.i ** 32)

obj = Power(2)
threading.Thread(target=obj.action).start()    # запуск связанного метода

# вложенная область видимости, для сохранения информации о состоянии
def action(i):
    def power():
        print(i ** 32)
    return power

threading.Thread(target=action(2)).start()     # запуск возвращаемой функции

# запуск обоих вариантов с помощью модуля _thread
_thread.start_new_thread(obj.action, ())       # запуск вызываемого объекта
_thread.start_new_thread(action(2), ())
```

Как видите, интерфейс модуля `threading` такой же гибкий, как и сам язык Python.

Еще раз о синхронизации доступа к совместно используемым объектам и переменным

Выше мы видели, что вызовы функции `print` в потоках выполнения необходимо синхронизировать с помощью блокировок, чтобы избежать смешивания выводимых данных, потому что стандартный поток вывода совместно используется всеми потоками выполнения. Строго говоря, потоки выполнения должны синхронизировать любые операции изме-

нения совместно используемых объектов и переменных. В зависимости от целей программы в число этих объектов могут входить:

- Изменяемые объекты в памяти (объекты, ссылки на которые передаются потокам или приобретаются каким-то иным способом, продолжительность существования которых превышает время работы потоков)
- Переменные в глобальной области видимости (изменяемые переменные, объявленные за пределами функций и классов, выполняемых в потоках)
- Содержимое модулей (для каждого модуля существует всего одна копия записи в системной таблице модулей)

Даже при работе с простыми глобальными переменными может потребоваться координация действий, если есть вероятность одновременных попыток их изменения, как показано в примере 5.12.

Пример 5.12. PP4E\System\Threads\thread-add-random.py

```
"выводит различные результаты при каждом запуске под Windows 7"

import threading, time
count = 0

def adder():
    global count
    count = count + 1  # изменяет глобальную переменную
    time.sleep(0.5)    # потоки выполнения совместно используют
    count = count + 1  # глобальные объекты и переменные

threads = []
for i in range(100):
    thread = threading.Thread(target=adder, args=())
    thread.start()
    threads.append(thread)

for thread in threads: thread.join()
print(count)
```

Этот пример порождает 100 потоков выполнения, каждый из которых дважды изменяет одну и ту же глобальную переменную (с задержкой между ними, чтобы обеспечить чередование операций в различных потоках). При каждом запуске в Windows 7 этот сценарий будет воспроизводить различные результаты:

```
C:\...\PP4E\System\Threads> thread-add-random.py
189

C:\...\PP4E\System\Threads> thread-add-random.py
200
```

```
C:\...\PP4E\System\Threads> thread-add-random.py
194

C:\...\PP4E\System\Threads> thread-add-random.py
191
```

Это объясняется тем, что потоки выполнения произвольно перекрываются друг с другом по времени: интерпретатор не гарантирует, что инструкции – даже такие простые инструкции присваивания, как в данном примере, – будут выполнены полностью до того, как управление перейдет другому потоку выполнения (то есть они не являются атомарными). Когда один поток изменяет значение глобальной переменной, он может получить промежуточный результат, произведенный другим потоком. Как следствие этого мы наблюдаем непредсказуемое поведение. Чтобы заставить этот сценарий работать корректно, необходимо снова воспользоваться блокировками для синхронизации изменений – в какой бы момент мы ни запускали сценарий из примера 5.13, он всегда будет выводить число 200.

Пример 5.13. PP4E\System\Threads\thread-add-synch.py

```
"всегда выводит 200 - благодаря синхронизации доступа к глобальному ресурсу"

import threading, time
count = 0

def adder(addlock):       # совместно используемый объект блокировки
    global count
    with addlock:         # блокировка приобретается/освобождается
        count = count + 1 # автоматически
    time.sleep(0.5)
    with addlock:         # в каждый конкретный момент времени
        count = count + 1 # только 1 поток может изменить значение переменной

addlock = threading.Lock()
threads = []
for i in range(100):
    thread = threading.Thread(target=adder, args=(addlock,))
    thread.start()
    threads.append(thread)

for thread in threads: thread.join()
print(count)
```

Некоторые простейшие операции в языке Python являются атомарными и не требуют синхронизации, тем не менее лучше все-таки предусматривать координацию потоков выполнения, если есть вероятность одновременных попыток изменения значения. Со временем может измениться не только набор атомарных операций, но и внутренняя реализация механизма потоков выполнения (такие изменения ожидаются в версии Python 3.2, как описывается далее).

Конечно, это во многом искусственный пример (порождать 100 потоков выполнения, чтобы в каждом из них дважды увеличить счетчик, – это определенно не самый практичный случай использования потоков!), но он наглядно иллюстрирует проблемы, с которыми можно столкнуться, когда существует вероятность параллельного изменения объекта или переменной, совместно используемой потоками. К счастью, для многих, если не для большинства применений, модуль `queue`, описываемый в следующем разделе, способен обеспечить автоматическую синхронизацию потоков выполнения.

Прежде чем двинуться дальше, я должен отметить, что помимо классов `Thread` и `Lock` в модуле `threading` имеются и другие высокоуровневые инструменты синхронизации доступа к совместно используемым объектам (например, `Semaphore`, `Condition`, `Event`) – много больше, чем позволяет вместить объем этой книги, поэтому за дополнительными подробностями обращайтесь к руководству по библиотеке. Дополнительные примеры использования потоков выполнения и дочерних процессов вы найдете в оставшейся части этой главы, а также среди примеров в разделах книги, посвященных реализации графического интерфейса и сетевых взаимодействий. В графических интерфейсах, например, мы будем использовать потоки, чтобы избежать их блокирования. Мы также будем порождать потоки и дочерние процессы в сетевых серверах, чтобы исключить вероятность отказа в обслуживании клиентов.

Кроме того, мы будем исследовать приемы использования модуля `threading` для завершения программы без применения метода `join`, но в соединении с очередями – которые являются темой следующего раздела.

Модуль queue

Синхронизировать доступ потоков выполнения к совместно используемым ресурсам можно с помощью блокировок, но часто в этом нет необходимости. Как уже упоминалось выше, на практике многопоточные программы часто организуются, как набор потоков производителей и потребителей, которые взаимодействуют между собой, помещая данные в общую очередь и извлекая их оттуда. Если очередь синхронизирует доступ к самой себе, она автоматически будет синхронизировать взаимодействия потоков выполнения.

Как раз такое хранилище данных реализует модуль `queue` из стандартной библиотеки. Он предоставляет стандартную очередь данных – список объектов Python, построенный по принципу «первый пришел, первый ушел» (first-in first-out, fifo), в котором добавление элементов производится с одного конца, а удаление – с другого. Подобно обычным спискам, очереди, реализуемые этим модулем, могут содержать объекты любых типов, включая объекты простых типов (строки, списки, словари и так далее) и более экзотических типов (экземпляры классов, произвольные вызываемые объекты, такие как функции и связанные методы, и многие другие).

Однако, в отличие от обычных списков, объект очереди автоматически управляет операциями приобретения и освобождения блокировки, благодаря чему в каждый конкретный момент времени изменять очередь может только один поток. Вследствие этого программы, использующие очереди для организации взаимодействий между потоками, изначально обеспечивают поддержку многопоточной модели выполнения и обычно не используют свои собственные блокировки для доступа к данным из потоков выполнения.

Подобно другим инструментам из арсенала поддержки потоков выполнения в языке Python, очереди удивительно просты в использовании. Так, сценарий в примере 5.14 порождает два потока-потребителя, которые ожидают появления данных в общей очереди, и четыре потока-производителя, периодически, через определенные интервалы времени, помещающие данные в очередь (для каждого из них установлена своя продолжительность интервала, чтобы имитировать выполнение длительных операций). Другими словами, эта программа запускает семь потоков выполнения (включая главный поток), шесть из которых обращаются к общей очереди параллельно.

Пример 5.14. PP4E\System\Threads\queuetest.py

```
"взаимодействие потоков производителей и потребителей посредством очереди"

numconsumers = 2       # количество потоков-потребителей
numproducers = 4       # количество потоков-производителей
nummessages  = 4       # количество сообщений, помещаемых производителем

import _thread as thread, queue, time
safeprint = thread.allocate_lock() # в противном случае вывод может
                                   # перемешиваться
dataQueue = queue.Queue()          # общая очередь неограниченного размера

def producer(idnum):
    for msgnum in range(nummessages):
        time.sleep(idnum)
        dataQueue.put('[producer id=%d, count=%d]' % (idnum, msgnum))

def consumer(idnum):
    while True:
        time.sleep(0.1)
        try:
            data = dataQueue.get(block=False)
        except queue.Empty:
            pass
        else:
            with safeprint:
                print('consumer', idnum, 'got =>', data)

if __name__ == '__main__':
```

```
    for i in range(numconsumers):
        thread.start_new_thread(consumer, (i,))
    for i in range(numproducers):
        thread.start_new_thread(producer, (i,))
    time.sleep(((numproducers-1) * nummessages) + 1)
    print('Main thread exit.')
```

Прежде чем я покажу вывод этого сценария, я хочу подчеркнуть некоторые важные моменты в этом программном коде.

Аргумент или глобальная переменная?

Обратите внимание, что ссылка на очередь сохраняется в глобальной переменной. Благодаря этому очередь может использоваться всеми порожденными потоками выполнения (все они выполняются в одном процессе и в одном глобальном пространстве имен). Потоки изменяют сам объект очереди, а не ссылку в переменной, поэтому они точно так же могли бы работать с очередью, если бы она передавалась, как аргумент функции, выполняемой в потоке. Очередь является совместно используемым объектом в памяти, и неважно, каким способом поток обретет ссылку на него (полную версию сценария, фрагмент которого представлен ниже, вы найдете в файле *queuetest2.py*, в дереве примеров):

```
dataQueue = queue.Queue()      # общий объект, неограниченный размер

def producer(idnum, dataqueue):
    for msgnum in range(nummessages):
        time.sleep(idnum)
        dataqueue.put('[producer id=%d, count=%d]' % (idnum, msgnum))

def consumer(idnum, dataqueue): ...

if __name__ == '__main__':
    for i in range(numproducers):
        thread.start_new_thread(producer, (i, dataQueue))
    for i in range(numproducers):
        thread.start_new_thread(producer, (i, dataQueue))
```

Завершение программы с дочерними потоками выполнения

Обратите также внимание, что сценарий завершает свою работу вместе с завершением главного потока, при том, что потоки-потребители продолжают выполнять свой бесконечный цикл. Этот прием прекрасно действует в Windows (и в большинстве других систем) – при использовании модуля `_thread` программа просто завершает свою работу вместе с главным потоком. Именно поэтому мы использовали функцию `sleep` в некоторых примерах – чтобы дать дочерним потокам возможность завершить свою работу, и именно поэтому нам нет необходимости беспокоиться о завершении потоков-потребителей, которые в данном примере выполняются в бесконечном цикле.

Однако при использовании альтернативного модуля `threading` программа не может завершиться, когда хотя бы один поток продолжает работу, если только он не был запущен, как *поток-демон*. В частности, программа завершается, когда в ней остаются только потоки-демоны. При создании потоки наследуют признак принадлежности к потокам-демонам от потока, породившего их. Главный поток в программах на языке Python не может быть демоном, тогда как потоки, созданные без помощи этого модуля, считаются демонами (включая некоторые потоки, создаваемые расширениями на языке C). Чтобы переопределить признак, унаследованный по умолчанию, можно вручную установить атрибут `daemon` объекта потока. Другими словами, потоки, не относящиеся к потокам-демонам, препятствуют завершению программы, и программы продолжают работать, пока не завершатся все потоки, созданные под управлением модуля `threading`.

Эту особенность можно рассматривать как достоинство или как недостаток, в зависимости от потребностей программы, – с одной стороны, когда не используется метод `join`, или когда главный поток не приостанавливается на некоторое время, она может принудительно завершать рабочие потоки; с другой стороны, она может препятствовать завершению программы, как показано в примере 5.14. Чтобы этот пример мог работать при использовании модуля `threading`, используйте следующее альтернативное решение (смотрите полную версию в файле *queuetest3.py* в дереве примеров, а также сценарий *thread-countthreading.py* – в качестве демонстрации того, где может пригодиться препятствование завершению):

```
import threading, queue, time

def producer(idnum, dataqueue): ...

def consumer(idnum, dataqueue): ...

if __name__ == '__main__':
    for i in range(numconsumers):
        thread = threading.Thread(target=consumer, args=(i, dataQueue))
        thread.daemon = True # иначе программа не завершится!
        thread.start()

    waitfor = []
    for i in range(numproducers):
        thread = threading.Thread(target=producer, args=(i, dataQueue))
        waitfor.append(thread)
        thread.start()

    for thread in waitfor: thread.join() # или большое значение в time.sleep()
    print('Main thread exit.')
```

Мы еще вернемся к потокам-демонам и к проблеме завершения потоков в главе 10, когда будем изучать особенности реализации графических

интерфейсов. Как мы увидим, в том контексте все происходит точно так же, за исключением того, что там главный поток обычно занимается обслуживанием графического интерфейса.

Запуск сценария

Теперь вернемся к примеру 5.14. Ниже приводится вывод этого примера после запуска на моем компьютере под управлением Windows. Обратите внимание, что несмотря на автоматическую координацию обмена данными между потоками с помощью очереди, в этом сценарии по-прежнему необходимо использовать блокировку для синхронизации доступа к стандартному потоку вывода – очередь синхронизирует обмен данными, но в некоторых программах все равно может потребоваться использовать блокировки для других целей. Как было показано в предыдущих примерах, если не использовать блокировку `safeprint`, вывод от разных потоков-потребителей может перемешиваться, поскольку есть вероятность, что поток-потребитель будет приостановлен в процессе выполнения операции вывода:

```
C:\...\PP4E\System\Threads> queuetest.py
consumer 1 got => [producer id=0, count=0]
consumer 0 got => [producer id=0, count=1]
consumer 1 got => [producer id=0, count=2]
consumer 0 got => [producer id=0, count=3]
consumer 1 got => [producer id=1, count=0]
consumer 1 got => [producer id=2, count=0]
consumer 0 got => [producer id=1, count=1]
consumer 1 got => [producer id=3, count=0]
consumer 0 got => [producer id=1, count=2]
consumer 1 got => [producer id=2, count=1]
consumer 1 got => [producer id=1, count=3]
consumer 1 got => [producer id=3, count=1]
consumer 0 got => [producer id=2, count=2]
consumer 1 got => [producer id=2, count=3]
consumer 1 got => [producer id=3, count=2]
consumer 1 got => [producer id=3, count=3]
Main thread exit.
```

Попробуйте поэкспериментировать с параметрами в начале этого сценария. Единственный потребитель, например, мог бы имитировать главный поток графического интерфейса. Ниже представлен вывод сценария с единственным потребителем – потоки-производители попрежнему добавляют данные в очередь в достаточно случайном порядке, потому что потоки выполняются параллельно друг с другом и с потоком-потребителем:

```
C:\...\PP4E\System\Threads> queuetest.py
consumer 0 got => [producer id=0, count=0]
consumer 0 got => [producer id=0, count=1]
consumer 0 got => [producer id=0, count=2]
consumer 0 got => [producer id=0, count=3]
```

```
consumer 0 got => [producer id=1, count=0]
consumer 0 got => [producer id=2, count=0]
consumer 0 got => [producer id=1, count=1]
consumer 0 got => [producer id=3, count=0]
consumer 0 got => [producer id=1, count=2]
consumer 0 got => [producer id=2, count=1]
consumer 0 got => [producer id=1, count=3]
consumer 0 got => [producer id=3, count=1]
consumer 0 got => [producer id=2, count=2]
consumer 0 got => [producer id=2, count=3]
consumer 0 got => [producer id=3, count=2]
consumer 0 got => [producer id=3, count=3]
Main thread exit.
```

Кроме основных особенностей очередей, продемонстрированных в этом сценарии, очереди могут иметь фиксированный или неограниченный размер, а методы `get` и `put` могут блокировать или не блокировать вызывающий поток – подробное описание интерфейса очередей вы найдете в руководстве по стандартной библиотеке Python. Однако, поскольку мы только что попробовали сымитировать типичную структуру сценария с графическим интерфейсом, продолжим исследование этого понятия дальше.

Графические интерфейсы и потоки выполнения: предварительное знакомство

Мы еще вернемся к потокам выполнения и очередям и рассмотрим дополнительные примеры их использования, когда позже будем изучать приемы создания графических интерфейсов. В примере приложения PyMailGUI, представленном в главе 14, широко будут использоваться инструменты управления потоками выполнения и очередями, представленными здесь. В главах 10 и 9 будут обсуждаться особенности использования многопоточной модели выполнения в контексте библиотеки tkinter инструментов для построения графического интерфейса, – как только мы познакомимся с ней поближе. В этом разделе мы не будем погружаться в программный код, но отметим, что потоки выполнения обычно являются неотъемлемой частью большинства нетривиальных графических интерфейсов. Модель функционирования многих графических интерфейсов представляет собой комбинацию потоков выполнения, очередей и циклов, выполняемых по таймеру.

И вот почему. В контексте графического интерфейса любая операция, выполнение которой может быть заблокировано или занять продолжительное время, должна запускаться в параллельном потоке, чтобы графический интерфейс (главный поток) оставался активным и продолжал откликаться на действия пользователя. Подобные операции можно было бы запускать в виде отдельных процессов, однако эффективность потоков выполнения и поддерживаемая ими возможность совместного использования памяти процесса делает их идеальным инструментом

для решения подобных задач. Кроме того, так как большая часть инструментов создания графических интерфейсов не позволяет обновлять интерфейс сразу из нескольких потоков, то внесение изменений в графический интерфейс будет ограничено главным потоком.

Так как только главный поток должен в общем случае изменять интерфейс, программы с графическим интерфейсом обычно принимают следующий вид: главный поток, который обслуживает интерфейс, и один или несколько долгоживущих потоков-производителей – по одному потоку для каждой задачи. Для синхронизации потоков все глобальные данные передаются между ними с помощью глобальных очередей: рабочие потоки посылают результаты, а поток, управляющий графическим интерфейсом, потребляет их.

Если говорить более определенно:

- *Главный поток* выполняет все изменения в графическом интерфейсе и запускает цикл, выполняющийся по таймеру, который выполняет периодическую проверку наличия новых данных в очереди для отображения на экране. В библиотеке tkinter, например, для периодической проверки очереди можно использовать метод `after(msecs, func, *args)`. Так как такие события распространяются процессором событий графического интерфейса, все изменения в интерфейсе будут выполняться только в главном потоке (и часто это является обязательным требованием, из-за того что инструменты создания графических интерфейсов редко поддерживают многопоточную модель выполнения).
- *Дочерние потоки* вообще не выполняют операций с графическим интерфейсом. Они лишь производят данные и помещают их в очередь, откуда эти данные будут извлекаться главным потоком. При необходимости дочерние потоки могут помещать в очередь функции обратного вызова, которые будут вызываться главным потоком. В программах с графическим интерфейсом обычно недостаточно просто передать функцию обратного вызова, изменяющую интерфейс, из главного потока в дочерний и вызывать ее оттуда – функция будет выполняться в дочернем потоке и, возможно, параллельно с другими потоками.

Так как потоки обеспечивают более высокую отзывчивость графического интерфейса, чем цикл на основе таймера, такая организация приложения позволяет избежать блокирования интерфейса (потоки-производители работают параллельно с графическим интерфейсом) и не терять входящие события (потоки-производители действуют независимо от цикла событий графического интерфейса и выполняются с максимальной скоростью). Главный поток, обслуживающий графический интерфейс, будет отображать результаты из очереди так быстро, как только это возможно в контексте более медленного цикла событий графического интерфейса.

Кроме того, имейте в виду, что, независимо от наличия поддержки многопоточной модели выполнения в инструменте создания графического интерфейса, многопоточные программы с графическим интерфейсом по-прежнему должны придерживаться общих принципов построения многопоточных программ – может потребоваться синхронизировать доступ к общим ресурсам, если он выходит за рамки модели с очередью, совместно используемой производителями/потребителями. Если дочерние потоки должны изменять другие общие ресурсы, используемые главным потоком графического интерфейса, может потребоваться использовать блокировки, чтобы избежать их взаимовлияния. Например, дочерние потоки, загружающие и кэширующие сообщения электронной почты, не должны перекрываться по времени с другими потоками, использующими или изменяющими содержимое кэша. То есть одних только очередей может оказаться недостаточно. Если обязанность потоков не сводится к размещению своих данных в очередь, то многопоточные приложения с графическим интерфейсом должны учитывать проблемы, сопутствующие параллельной обработке данных.

Далее мы увидим, как можно реализовать многопоточную модель в графическом интерфейсе. Дополнительную информацию по этой теме вы найдете в дискуссии, посвященной использованию потоков выполнения при работе с инструментом tkinter создания графических интерфейсов в главе 9, в примерах реализации инструментов для работы с потоками выполнения и очередями в главе 10 и в примере приложения PyMailGUI в главе 14.

Далее в этой главе мы также встретимся с пакетом `multiprocessing`, поддержка процессов и очередей в котором предоставляет новые возможности реализации модели графического интерфейса, где вместо потоков выполнения используются процессы. Эта модель позволяет обойти ограничение GIL, но ее применение может отрицательно сказываться на производительности, в зависимости от платформы, и может оказаться вообще неприменимой в контексте потоков (эта модель не поддерживает прямой доступ к общим изменяемым объектам, хранящим информацию о состоянии потоков выполнения, однако она поддерживает механизм обмена сообщениями). Но сначала рассмотрим несколько интересных моментов, касающихся потоков.

Таймеры-потоки против таймеров графического интерфейса

Интересно отметить, что модуль `threading` экспортирует универсальную функцию `Timer`, которая, как и метод `after` виджетов в библиотеке tkinter, может использоваться для запуска другой функции по истечении указанного интервала времени:

```
Timer(N.M, somefunc).start() # вызовет функцию somefunc через N.M секунд
```

Объекты-таймеры имеют метод `start()`, запускающий таймер, а также метод `cancel()`, позволяющий отменить запланированное событие, а кроме того, ожидание в них реализовано в виде отдельного потока выполнения. Например, следующий пример выведет сообщение спустя 5.5 секунд:

```
>>> import sys
>>> from threading import Timer
>>> t = Timer(5.5, lambda: print('Spam!')) # дочерний поток
>>> t.start()
>>> Spam!
```

Этот инструмент может пригодиться в самых разных ситуациях, но он не должен использоваться в графических интерфейсах: отложенная функция будет вызвана в контексте дочернего потока, а не в главном потоке графического интерфейса, поэтому она не должна выполнять изменения в графическом интерфейсе. Метод `after` из библиотеки tkinter, напротив, вызывается из цикла обработки событий главного потока выполнения и запускает указанную функцию в главном потоке, поэтому она свободно может изменять графический интерфейс.

Например, следующий пример выведет окно диалога через 5.5 секунд в контексте главного потока инструмента tkinter (в некоторых интерфейсах вам может также потребоваться запустить `win.main loop()`):

```
>>> from tkinter import Tk
>>> from tkinter.messagebox import showinfo
>>> win = Tk()
>>> win.after(5500, lambda: showinfo('Popup', 'Spam!'))
```

В последней строке здесь планируется однократный запуск функции в главном потоке выполнения графического интерфейса, но он не приостанавливает работу вызывающей программы и поэтому не блокирует графический интерфейс. Ниже приводится эквивалентная и более простая форма:

```
>>> win.after(5500, showinfo, 'Popup', 'Spam')
```

В следующей части книги, в главе 9, подробнее будет рассказываться о библиотеке tkinter и о методе `after`, а в главе 10 – о роли потоков выполнения в приложениях с графическим интерфейсом.

Подробнее о глобальной блокировке интерпретатора (GIL)

Эта тема относится к области низкоуровневого программирования, и во многих случаях можно и без этих знаний успешно организовать многопоточную работу в программах на языке Python, но тем не менее реализация механизма потоков в Python может оказывать влияние как на производительность, так и на стиль программирования. В этом разделе приводятся сведения об особенностях реализации и о некоторых их следствиях.

О реализации потоков выполнения в грядущей версии Python 3.2: В этом разделе описывается текущая реализация потоков выполнения, включая версию Python 3.1. К моменту написания этих строк версия Python 3.2 все еще находилась в стадии разработки, но одним из нововведений в ней наверняка будет новая версия GIL, обеспечивающая более высокую производительность, особенно в системах с многоядерными процессорами. Новая реализация GIL по-прежнему будет синхронизировать доступ к PVM (программный код на языке Python по-прежнему будет мультиплексироваться, как и ранее), но она будет использовать более эффективную схему переключения контекста, чем ныне используемая схема переключения через N-инструкций-в-байткоде.

Помимо всего прочего, текущая функция `sys.setcheckinterval`, вероятно, будет заменена таймером с поддержкой новой схемы. В частности, понятие интервала проверки необходимости переключения потоков выполнения будет ликвидировано и заменено абсолютным значением продолжительности, выраженным в секундах. Как ожидается, это значение по умолчанию будет равно 5 миллисекундам, но его можно будет изменять с помощью функции `sys.setswitchinterval`.

Кроме того, существовало множество планов по полной ликвидации GIL (включая проект Unladen Swallow, запущенный сотрудниками Google), однако до сих пор не было представлено ни одного варианта решения. Я не берусь предсказывать будущее, поэтому читайте документацию к новым версиям Python, чтобы оставаться в курсе.

Строго говоря, в настоящее время Python использует механизм *глобальной блокировки интерпретатора* (Global Interpreter Lock, GIL), представленный в начале этого раздела и обеспечивающий выполнение интерпретатором Python в каждый конкретный момент времени программного кода не более чем одного потока. Кроме того, чтобы дать каждому потоку возможность поработать, интерпретатор автоматически переключается между ними через равные промежутки времени (в Python 3.1 – путем освобождения и приобретения блокировки после выполнения некоторого числа инструкций в байт-коде), а также в на-

чале длительных операций (например, в начале операций ввода/вывода в файлы).

Такая схема позволяет избежать проблем, могущих возникнуть, когда нескольким потокам выполнения одновременно может потребоваться обновить системные данные Python. Например, если двум потокам разрешить одновременно изменить счетчик ссылок на объект, результаты могут оказаться непредсказуемыми. Применение этой схемы может также повлечь тонкие неочевидные последствия. В частности, в примерах использования потоков, приведенных в данной главе, мы видели, что при выводе в `stdout` текст может повреждаться, если потоки не синхронизируют свои операции вывода с помощью блокировок.

Кроме того, хотя глобальная блокировка интерпретатора не разрешает одновременно выполнять более одного потока Python, этого недостаточно для обеспечения безопасности потоков в целом и это никак не решает проблемы синхронизации на более высоком уровне. Например, если одновременно несколько потоков пытаются *изменить* одну и ту же переменную, им обычно должен предоставляться исключительный доступ к объекту с помощью блокировок. В противном случае существует вероятность, что переключение потоков произойдет посреди байт-кода выражения, осуществляющего изменение.

Не обязательно использовать блокировки для обращения ко всем совместно используемым объектам, особенно если только один поток изменяет объект, а остальные только наблюдают за изменениями. Возьмите за правило всегда использовать блокировки для синхронизации потоков в тех случаях, когда возможна конкуренция в операции изменения, не полагаясь на текущий способ реализации потоков.

Интервал переключения потоков выполнения

В некоторых случаях параллельные изменения могут выполняться корректно и без применения блокировок, если сделать интервал переключения потоков настолько большим, чтобы каждый из потоков мог выполниться прежде чем будет переключен. Функция `sys.setcheckinterval(N)` устанавливает частоту, с которой интерпретатор будет выполнять такие операции, как переключение потоков и обработка сигналов.

Этот интервал измеряется в инструкциях 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: уменьшатся накладные расходы на переключение потоков, но потоки медленнее будут реагировать на события. Установка более низких значений обеспечит более высокую отзывчивость потоков на события, но увеличит накладные расходы на их переключение.

Атомарные операции

Из-за того, каким образом интерпретатор Python использует GIL для синхронизации доступа к виртуальной машине, ни для каких инструкций высокого уровня не гарантируется выполнение инструкции до конца до переключения на другой поток, но оно гарантируется для всех инструкций в байт-коде. Инструкции в байт-коде являются неделимыми, поэтому некоторые операции в языке Python обеспечивают безопасную работу с потоками. Такие операции называются *атомарными*, потому что их выполнение не может быть прервано – и при их использовании не требуется задействовать блокировки или очереди, чтобы избежать проблем, связанных с одновременными изменениями. Например, к моменту написания этих строк в стандартной реализации C Python выполняются атомарно: метод `list.append`, операции извлечения и некоторые операции присваивания значений переменным, обращение к элементам списков, ключам словарей и атрибутам объектов, а также некоторые другие операции. Другие операции, такие как `x = x+1` (и вообще любые операции, при выполнении которых происходит чтение данных, их изменение и запись обратно), – нет.

Однако, как уже отмечалось выше, не следует полагаться на эти особенности, потому что они требуют глубокого понимания внутренней реализации интерпретатора и могут изменяться от версии к версии. На самом деле, набор атомарных операций может существенно измениться с введением более свободной от ограничений реализации потоков. На практике проще использовать блокировки для доступа ко всем глобальным переменным и общим объектам, чем пытаться запомнить, какие операции могут или не могут быть безопасными при одновременном использовании их в нескольких потоках выполнения.

Прикладной интерфейс потоков на языке C

Наконец, если вы собираетесь использовать смешанный программный код на языках Python и C, посмотрите также интерфейсы потоков, описываемые в стандартном руководстве по API Python/C. В многопоточных программах расширения на языке C должны освобождать и снова приобретать глобальную блокировку интерпретатора при выполнении длительных операций, чтобы позволить выполняться другим потокам Python. В частности, функции в расширениях на языке C, выполняющие продолжительные операции, должны освобождать блокировку на входе и приобретать на выходе, чтобы возобновить работу программного кода Python.

Обратите внимание: хотя программный код Python в разных потоках Python не может выполняться одновременно из-за синхронизации с помощью GIL, тем не менее фрагменты потоков с программным кодом на языке C такую возможность имеют. Параллельно может выполняться

любое число потоков, при условии, что они действуют за пределами виртуальной машины Python. Потоки на языке C могут перекрываться во времени и с другими потоками на языке C, и с потоками Python, выполняемыми виртуальной машиной. Благодаря этому разделение программного кода по библиотекам на языке C может применяться в приложениях Python для использования преимуществ многопроцессорных систем.

Однако часто бывает проще использовать преимущества многопроцессорных систем за счет создания программ на языке Python, которые вместо потоков запускают параллельные процессы. Сложность программного кода, управляющего потоками и процессами, примерно одинаковая. Подробнее о расширениях на языке C и их требованиях к многопоточной модели выполнения рассказывается в главе 20. Тем не менее коротко отмечу, что в состав Python входят инструменты на языке C (среди них пара макроопределений для управления GIL), которые могут использоваться для обертывания продолжительных операций в программном коде расширений на языке C и позволяют параллельно выполняться другим потокам в программном коде на языке Python.

Альтернатива на основе процессов: пакет multiprocessing (описывается далее)

К настоящему моменту у вас должно сложиться общее представление о параллельно выполняющихся процессах и потоках, а также об инструментах в языке Python для управления ими. Далее в главе мы вернемся к этим идеям, когда будем знакомиться с пакетом `multiprocessing` – инструментом из стандартной библиотеки, объединяющим в себе простоту и переносимость потоков с преимуществами процессов, – за счет реализации прикладного интерфейса, напоминающего потоки, который вместо потоков запускает процессы. Он стремится решить проблемы переносимости поддержки процессов и ограничений на использование преимуществ многопроцессорных систем, накладываемых блокировкой GIL. Но в некоторых ситуациях он не может использоваться как замена приему ветвления процессов и накладывает ряд ограничений, которые отсутствуют при работе с потоками, проистекающих из особенностей модели процессов (например, изменяемые объекты не могут использоваться непосредственно, потому что их приходится копировать через границы процессов, а объекты, не поддерживающие возможность сериализации, такие как связанные методы, вообще не могут использоваться).

Пакет `multiprocessing` реализует набор инструментов, упрощающих такие задачи, как взаимодействие между процессами и передача кода завершения. Поэтому мы сначала исследуем поддержку этих возможностей в языке Python и попутно рассмотрим еще несколько примеров использования потоков выполнения и процессов.

Завершение программ

Как мы видели выше, в отличие от языка C, в Python нет функции «main». При запуске программы весь программный код верхнего уровня в файле (то есть в файле, имя которого указано в командной строке, на котором был выполнен щелчок в проводнике, и так далее) просто выполняется от начала и до конца. Обычно сценарии завершаются, когда интерпретатор достигает конца файла, но завершить программу можно и явно с помощью инструментов из модулей `sys` и `os`.

Завершение программ средствами модуля sys

Например, программу можно завершить раньше обычного, вызвав функцию `sys.exit`:

```
>>> sys.exit(N) # выход с кодом завершения N, в противном случае
                # программа завершится по достижении конца сценария
```

Интересно отметить, что в действительности эта функция просто возбуждает встроенное исключение `SystemExit`. Поэтому его можно обычным образом перехватывать, чтобы выполнить завершающие действия. Если это исключение не перехватывать, интерпретатор завершит работу как обычно. Например:

```
C:\...\PP4E\System> python
>>> import sys
>>> try:
...     sys.exit()             # смотрите также: os._exit, Tk().quit()
... except SystemExit:
...     print('ignoring exit')
...
ignoring exit
>>>
```

Некоторые программные инструменты, такие как отладчики, могут использовать эту особенность для предотвращения завершения программы. Фактически явное возбуждение встроенного исключения `SystemExit` с помощью инструкции `raise` эквивалентно вызову функции `sys.exit`. В практических сценариях в блоке `try` можно было бы перехватывать исключения завершения работы, возбуждаемые в любом месте программы. Сценарий в примере 5.15 завершается из выполняющейся функции.

Пример 5.15. PP4E\System\Exits\testexit_sys.py

```
def later():
    import sys
    print('Bye sys world')
    sys.exit(42)
    print('Never reached')

if __name__ == '__main__': later()
```

Если запустить этот пример как самостоятельный сценарий, он завершится еще до того, как интерпретатор достигнет конца файла. Но поскольку функция sys.exit возбуждает исключение, в случае импортирования этой функции вызывающий программный код может перехватывать возбуждаемое исключение завершения и отменять его, либо предусматривать блок finally, который будет выполнен при завершении программы:

```
C:\...\PP4E\System\Exits> python testexit_sys.py
Bye sys world

C:\...\PP4E\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:\...\PP4E\System\Exits> # процесс интерактивного сеанса завершился
```

Завершение программ средствами модуля os

Можно выйти из Python и другими способами. Например, в дочернем процессе в Unix обычно вызывается функция os._exit, а не sys.exit. Потоки можно завершать с помощью функции _thread.exit, а приложения с графическим интерфейсом на основе tkinter часто завершаются с помощью метода Tk().quit (). С модулем tkinter мы познакомимся далее в этой книге, а сейчас поближе рассмотрим инструменты завершения программ в модуле os.

При вызове функции os._exit вызывающий процесс завершается сразу, не возбуждая исключения, которое можно перехватить и игнорировать. Фактически при таком завершении процесс прекращает работу, не выталкивая буферы потоков вывода и не вызывая обработчики, выполняющие заключительные операции (которые можно определить с помощью модуля atexit из стандартной библиотеки), поэтому в общем случае данная функция должна использоваться только дочерними процессами, когда не требуется выполнения действий по завершению всей программы. Пример 5.16 иллюстрирует основы использования этой функции.

Пример 5.16. PP4E\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`**, функция** `os._exit` **неуязвима для инструкций обработки исключений** `try/except` **и** `try/finally`**:**

```
C:\...\PP4E\System\Exits> python testexit_os.py
Bye os world

C:\...\PP4E\System\Exits> python
>>> from testexit_os import outahere
>>> try:
...     outahere()
... except:
...     print('Ignored')
...
Bye os world # завершение процесса интерактивного сеанса

C:\...\PP4E\System\Exits> python
>>> from testexit_os import outahere
>>> try:
...     outahere()
... finally:
...     print('Cleanup')
...
Bye os world # ditto
```

Коды завершения команд оболочки

Обе функции завершения из модулей `sys` **и** `os`**, с которыми мы только что познакомились, принимают аргумент, определяющий код завершения процесса (в функции из модуля** `sys` **он необязателен, но в функции из модуля** `os` **– необходим). После завершения программы этот код может запрашиваться оболочкой или программой, запустившей сценарий как дочерний процесс. В Linux, например, чтобы получить код завершения последней программы, запрашивается значение переменной оболочки** `status`**. По соглашению ненулевое значение указывает, что возникли какие-то проблемы:**

```
[mark@linux]$ python testexit_sys.py
Bye sys world
[mark@linux]$ echo $status
42
[mark@linux]$ python testexit_os.py
Bye os world
```

```
[mark@linux]$ echo $status
99
```

В последовательности команд попутная проверка кодов завершения может использоваться как простая форма связи между программами.

Можно также получить код завершения программы, запущенной другим сценарием. Например, как рассказывалось в главах 2 и 3, при запуске команд оболочки код завершения предоставляется как:

- Значение, возвращаемое функцией `os.system`
- Значение, возвращаемое методом `close` объекта `os.popen` (по историческим причинам для значения `None` возвращается код `0`, что означает отсутствие ошибок)
- Значение, возвращаемое различными интерфейсами в модуле `subprocess` (например, возвращаемое значение функции `call`, значение атрибута `returnvalue` объекта `Popen` и возвращаемое значение метода `wait`)

Кроме того, в случае, когда программа запускается приемом ветвления процессов, код завершения можно получить вызовом функций `os.wait` и `os.waitpid` в родительском процессе.

Получение кода завершения с помощью os.system и os.popen

Рассмотрим сначала случай с командами оболочки – в операционной системе Linux запускаются программы из примеров 5.15 и 5.16, производится чтение вывода этих сценариев через каналы и получение кодов завершения:

```
[mark@linux]$ 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                  # извлекает код завершения из битовой маски
42

>>> pipe = os.popen('python testexit_os.py')
>>> stat = pipe.close()
>>> stat, stat >> 8
(25344, 99)
```

В версии Cygwin Python под Windows этот пример действует точно так же. При использовании функции `os.popen` в Unix-подобных системах по причинам, которые мы не будем здесь рассматривать, код завершения помещается в определенные битовые позиции возвращаемого значения.

Код действительно находится там, но чтобы его увидеть, нужно сдвинуть результат вправо на восемь разрядов. Код завершения команд, выполняемых с помощью функции os.system, можно получить с помощью библиотечной функции:

```
>>> stat = os.system('python testexit_sys.py')
Bye sys world
>>> stat, stat >> 8
(10752, 42)

>>> stat = os.system('python testexit_os.py')
Bye os world
>>> stat, stat >> 8
(25344, 99)
```

Все эти приемы действуют и в стандартной версии Python для Windows, однако в этой операционной системе код завершения уже не является битовой маской (проверяйте значение sys.platform, если ваша программа должна работать на обеих платформах):

```
C:\...\PP4E\System\Exits> python
>>> os.system('python testexit_sys.py')
Bye sys world
42
>>> os.system('python testexit_os.py')
Bye os world
99

>>> pipe = os.popen('python testexit_sys.py')
>>> pipe.read()
'Bye sys world\n'
>>> pipe.close()
42
>>>
>>> os.popen('python testexit_os.py').close()
99
```

Буферизация потока вывода: первый взгляд

Обратите внимание, что в последней проверке, в предыдущем фрагменте программного кода, не предпринимается попытка прочитать вывод команды. В подобных ситуациях может потребоваться запускать целевой сценарий в *небуферизованном* режиме, то есть запускать интерпретатор Python с флагом -u, или изменить сценарий, чтобы он выталкивал выходной буфер вручную с помощью функции sys.stdout.flush. В противном случае текст, выводимый в стандартный поток вывода, не будет вытолкнут из буфера стандартного потока вывода при вызове функции os._exit. По умолчанию при подключении канала, как в данном примере, стандартный поток вывода работает в режиме полной буферизации – при подключении к терминалу в буфер помещается только одна строка:

```
>>> pipe = os.popen('python testexit_os.py')
>>> pipe.read() # буферы не выталкиваются при выходе
''

>>> pipe = os.popen('python -u testexit_os.py') # принудительный
>>> pipe.read()                                 # небуферизованный режим
'Bye os world\n'
```

Странно, но, несмотря на то, что имеется возможность передавать функциям `os.popen` и `subprocess.Popen` аргумент, управляющий режимом и буферизацией, – в данном случае это не поможет. Аргументы передаются инструментам со стороны вызывающего процесса, с конца канала, работающего в порожденной программе, как поток ввода, а *не* как поток вывода:

```
>>> pipe = os.popen('python testexit_os.py', 'r', 1) # построчная буферизация
>>> pipe.read()              # но мой канал - это не поток вывода программы!
''

>>> from subprocess import Popen, PIPE
>>> pipe = Popen('python testexit_os.py', bufsize=1, stdout=PIPE) # для моего
>>> pipe.stdout.read()                                            # канала -
b''                                                               # не поможет
```

Аргументы, определяющие режим буферизации, воздействуют на поток вывода вызывающего процесса, через который он записывает данные в стандартный поток ввода команды, а не на поток вывода команды, откуда вызывающий процесс читает данные.

При необходимости запускаемый сценарий может сам, вручную выталкивать выходные буферы – периодически или перед принудительным завершением. Подробнее о буферизации мы поговорим, когда будем обсуждать возможность возникновения ситуации *взаимоблокировки* далее в этой главе, и еще раз – в главах 10 и 12, где мы узнаем, как все это увязывается с сокетами. Поскольку мы вспомнили про модуль `subprocess`, рассмотрим теперь предоставляемые им инструменты завершения программ.

Получение кода завершения с помощью модуля subprocess

Модуль `subprocess` позволяет получить код завершения различными способами, как было показано в главах 2 и 3 (значение `None` в атрибуте `returncode` указывает, что дочерний процесс еще не завершился):

```
C:\...\PP4E\System\Exits> python
>>> from subprocess import Popen, PIPE, call
>>> pipe = Popen('python testexit_sys.py', stdout=PIPE)
>>> pipe.stdout.read()
b'Bye sys world\r\n'
>>> pipe.wait()
42
```

```
>>> call('python testexit_sys.py')
Bye sys world
42

>>> pipe = Popen('python testexit_sys.py', stdout=PIPE)
>>> pipe.communicate()
(b'Bye sys world\r\n', None)
>>> pipe.returncode
42
```

Модуль subprocess действует аналогично и на Unix-подобных платформах, таких как Cygwin, но в отличие от функции os.popen, код завершения не преобразуется в битовую маску, и поэтому он совпадает с результатом в Windows (обратите внимание, что при использовании в Cygwin и в Unix-подобных системах требуется установить аргумент shell=True, как мы узнали в главе 2, тогда как в Windows этот аргумент требуется установить только для запуска встроенных команд оболочки, таких как dir):

```
[C:\...\PP4E\System\Exits]$ python
>>> from subprocess import Popen, PIPE, call
>>> pipe = Popen('python testexit_sys.py', stdout=PIPE, shell=True)
>>> pipe.stdout.read()
b'Bye sys world\n'
>>> pipe.wait()
42
>>> call('python testexit_sys.py', shell=True)
Bye sys world
42
```

Код завершения процесса и совместно используемая информация

Теперь, чтобы узнать, как получить код завершения процесса, порожденного ветвлением, напишем простую программу, выполняющую ветвление: сценарий в примере 5.17 порождает дочерние процессы и выводит коды их завершения, возвращаемые функцией os.wait, пока не будет нажата клавиша q".

Пример 5.17. PP4E\System\Exits\testexit_fork.py

```
"""
порождает дочерние процессы и получает коды их завершения вызовом функции
os.wait; прием ветвления может использоваться в Unix и Cygwin, но он не работает
в стандартной версии Python 3.1 для Windows;
примечание: порождаемые потоки выполнения совместно используют глобальные
переменные, но каждый процесс имеет собственные копии этих переменных (однако
при ветвлении процессов файловые дескрипторы используются совместно) --
exitstat здесь всегда имеет одно и то же значение, но может отличаться в случае
использования потоков;
"""
```

```
import os
exitstat = 0

def child():                    # здесь можно вызвать os.exit для завершения
    global exitstat             # изменит глобальную переменную этого процесса
    exitstat += 1               # код завершения для функции wait родителя
    print('Hello from child', os.getpid(), exitstat)
    os._exit(exitstat)
    print('never reached')

def parent():
    while True:
        newpid = os.fork()      # запустить новую копию процесса
        if newpid == 0:         # если это копия, вызвать функцию child
            child()             # ждать ввода 'q' с консоли
        else:
            pid, status = os.wait()
            print('Parent got', pid, status, (status >> 8))
            if input() == 'q': break

if __name__ == '__main__': parent()
```

Если запустить эту программу в Linux, Unix или Cygwin (не забывайте, что функция `fork` не работает в стандартной версии Python для Windows, – по крайней мере, когда я работал над четвертым изданием этой книги), она выведет следующие результаты:

```
[C:\...\PP4E\System\Exits]$ python testexit_fork.py
Hello from child 5828 1
Parent got 5828 256 1

Hello from child 9540 1
Parent got 9540 256 1

Hello from child 3152 1
Parent got 3152 256 1
q
```

Если внимательно изучить этот вывод, можно заметить, что код завершения (последнее выводимое число) всегда одинаков – 1. Поскольку ответвленные процессы начинают жизнь как *копии* создавших их процессов, они также получают копию глобальной памяти. Поэтому каждый дочерний процесс получает и изменяет собственную глобальную переменную `exitstat`, не трогая экземпляров этой переменной в других процессах. В то же время дочерние процессы получают копии файловых дескрипторов, которые используются совместно с родительским процессом, и именно поэтому вывод от дочерних процессов попадает в то же самое место.

Код завершения потока и совместно используемая информация

В отличие от процессов, потоки выполняются параллельно внутри одного и того же процесса и совместно используют глобальную память. Все потоки в примере 5.18 изменяют одну и ту же глобальную переменную exitstat.

Пример 5.18. PP4E\System\Exits\testexit_thread.py

```
"""
порождает потоки выполнения и следит за изменениями в глобальной памяти; обычно
потоки завершаются при возврате из выполняемой в них функции, но поток может
завершиться, вызвав функцию _thread.exit(); функция _thread.exit играет ту
же роль, что и функция sys.exit, и возбуждает исключение SystemExit; потоки
взаимодействуют через глобальные переменные, по мере надобности блокируемые;
ВНИМАНИЕ: на некоторых платформах может потребоваться придать атомарность
вызовам функций print/input -- из-за совместно используемых потоков ввода-
вывода;
"""

import _thread as thread
exitstat = 0

def child():
    global exitstat            # используется глобальная переменная процесса,
    exitstat += 1              # совместно используемая всеми потоками
    threadid = thread.get_ident()
    print('Hello from child', threadid, exitstat)
    thread.exit()
    print('never reached')

def parent():
    while True:
        thread.start_new_thread(child, ())
        if input() == 'q': break

if __name__ == '__main__': parent()
```

Ниже показано, как действует этот сценарий в Windows, – в отличие от ветвления процессов, потоки выполнения поддерживаются и в стандартной версии Python для Windows. Все потоки получают разные идентификаторы – они произвольные, но уникальные среди активных потоков, поэтому их можно использовать в качестве ключей словаря для сохранения информации о потоках (на некоторых платформах идентификаторы потоков могут повторно использоваться после их завершения):

```
C:\...\PP4E\System\Exits> python testexit_thread.py
Hello from child 4908 1

Hello from child 4860 2
```

```
Hello from child 2752 3

Hello from child 8964 4
q
```

Обратите внимание, что значение глобальной переменной exitstat в этом сценарии изменяется каждым потоком выполнения, – из-за того, что потоки совместно используют глобальную память процесса. Эта особенность часто используется для организации взаимодействий между потоками. Вместо того чтобы возвращать коды завершения, потоки могут присваивать значения глобальным переменным модуля или модифицировать изменяемые объекты, а для синхронизации доступа к совместно используемым элементам они могут использовать блокировки и очереди, если это необходимо. В данном сценарии также может возникнуть потребность в синхронизации потоков для изменения глобального счетчика, если он когда-либо будет использоваться для решения практических задач. Может потребоваться синхронизировать даже обращения к функциям print и input, если на используемой платформе потоки могут одновременно обращаться к потокам ввода-вывода. В этом простом демонстрационном сценарии мы отказались от использования блокировок, предположив, что потоки не будут обращаться к этим операциям одновременно.

Как мы уже знаем, работа потока завершается нормальным образом и без сообщений, когда происходит возврат из функции, запущенной потоком, и значение, возвращаемое функцией, игнорируется. Кроме того, может быть вызвана функция _thread.exit для завершения вызвавшего ее потока явно и тихо. Эта функция действует почти в точности как sys.exit (но не принимает аргумента с кодом завершения) и возбуждает исключение SystemExit в вызвавшем ее потоке. Поэтому поток можно также досрочно завершить, вызвав функцию sys.exit или непосредственно возбудив исключение SystemExit. Следите, однако, за тем, чтобы не вызвать внутри функции потока функцию os._exit, – это может привести к странным результатам (на моей системе Linux в результате подвешивался весь процесс, а в Windows уничтожались все потоки процесса!).

В альтернативном модуле threading реализация потоков не имеет метода, эквивалентного функции _thread.exit(), но, так как единственное действие, которое выполняет последний, – это возбуждение исключения SystemExit, применение этой операции при использовании модуля threading даст такой же эффект – поток немедленно и тихо завершит работу, как, например, в следующем фрагменте (этот программный код находится в файле *testexit-threading.py* в дереве примеров):

```
import threading, sys, time

def action():
    sys.exit() # или возбуждение исключения SystemExit()
    print('not reached')
```

```
threading.Thread(target=action).start()
time.sleep(2)
print('Main exit')
```

Помните также, что потоки выполнения и процессы имеют собственные модели продолжительности жизни, которые мы исследовали выше. Напомним, что если дочерние потоки продолжают выполняться, то поведение, обеспечиваемое двумя модулями работы с потоками, будет различаться – на большинстве платформ программа завершится, если главный поток был создан с помощью инструментов модуля `_thread`, но не сможет завершиться, если использовался модуль `threading` и все дочерние потоки не были запущены, как потоки-демоны. В случае использования процессов является нормальным, когда дочерние процессы «переживают» своего родителя. Эту отличительную черту процессов легко объяснить, если помнить, что потоки выполнения – это всего лишь вызовы функций внутри процесса, а процессы – это более автономные и независимые единицы.

При правильном применении код завершения можно использовать для обнаружения ошибок и в простых протоколах обмена данными в системах, образуемых сценариями командной строки. Но при этом следует подчеркнуть, что в большинстве сценариев завершение совпадает с достижением конца исходного файла, а большинство функций потоков выполнения просто возвращают управление – явное завершение обычно предусматривается только для исключительных ситуаций и только в заданных контекстах. Взаимодействие между программами обычно обеспечивается более богатым набором инструментов, чем просто передача целочисленных кодов завершения, а каким – рассказывается в следующем разделе.

Взаимодействия между процессами

Как мы видели выше, когда сценарии порождают *потоки выполнения* – задачи, выполняемые параллельно внутри программы, – потоки могут естественным образом поддерживать связь друг с другом путем изменения и чтения переменных и объектов в совместно используемой глобальной памяти. Сюда относятся как доступные переменные и атрибуты, так и изменяемые объекты. Мы видели также, что следует позаботиться об использовании блокировок для синхронизации доступа к совместно используемым объектам, если есть вероятность одновременного их изменения из разных потоков. Потоки выполнения предлагают достаточно простую модель взаимодействий, и модуль `queue` во многих ситуациях реализует синхронизацию практически автоматически.

Все становится намного сложнее, когда сценарии запускают дочерние процессы и программы, вообще не имеющие совместно используемой памяти. Если определить виды взаимодействий, которые могут осу-

ществляться между программами, то окажется, что большинство вариантов мы уже рассмотрели в этой и в предыдущих главах. Например, ниже перечислены простые механизмы, которые могут рассматриваться, как инструменты взаимодействий между программами:

- Простые файлы
- Аргументы командной строки
- Коды завершения программ
- Переадресация стандартных потоков ввода-вывода
- Каналы, создаваемые с помощью функции `os.popen` и модуля `subprocess`

Например, передача параметров в командной строке и запись в потоки ввода позволяет передавать параметры выполнения программ; чтение потоков вывода и кодов завершения дает возможность получать результаты. Поскольку порождаемыми программами наследуются значения переменных окружения, их также можно рассматривать, как один из способов передачи контекста. Каналы, создаваемые при помощи функции `os.popen` или модуля `subprocess`, позволяют организовать еще более динамические взаимодействия: данные могут передаваться между программами в произвольные моменты времени, не только во время запуска или завершения.

Помимо этих механизмов в библиотеке Python есть и другие средства организации взаимодействий между процессами (Inter-Process Communication, IPC). К ним относятся сокеты, разделяемая память, сигналы, анонимные и именованные каналы и другие. Некоторые из них являются более переносимыми, некоторые менее переносимыми, и все они различаются по сложности и сфере использования. Например:

- *Сигналы* позволяют программам передавать простые уведомления другим программам.
- *Анонимные каналы* позволяют обмениваться данными потокам выполнения и родственным процессам, совместно использующим файловые дескрипторы, но этот механизм опирается на модель ветвления процессов в Unix-подобных системах, которая не является переносимой.
- *Именованные каналы* отображаются в файловую систему – они позволяют обмениваться данными полностью независимым программам, но они доступны в Python не на всех платформах.
- *Сокеты* отображаются на общесистемный набор номеров сетевых портов – они точно так же позволяют организовать обмен данными между произвольными программами, действующими на одном компьютере, но при этом предоставляют возможность взаимодействий по сети с программами, выполняющимися на удаленном компьютере, и к тому же представляют собой наиболее переносимый вариант.

Хотя некоторые из них могут использоваться, как механизмы взаимодействий между потоками выполнения, но истинная их мощь становится видна, когда они используются для организации взаимодействий между отдельными процессами, вообще не имеющими общей памяти.

В данном разделе мы познакомимся с каналами (анонимными и именованными), а также с сигналами. Помимо этого здесь мы впервые встретимся с сокетами, но только в виде предварительного знакомства. Сокеты могут использоваться для организации взаимодействий процессов, выполняющихся на одном компьютере, но так как основное их назначение заключается в работе с сетями, большую часть подробностей мы оставим до части книги, где будет рассказываться о разработке приложения для Интернета.

Программисты на языке Python могут пользоваться и другими механизмами IPC (например, разделяемой памятью, доступ к которой предоставляется модулем `mmap`), о которых здесь не рассказывается из-за недостатка места. Если вас интересует что-то более специальное, ищите в руководствах Python и на веб-сайте подробности использования других схем IPC.

По окончании этого раздела мы также исследуем пакет `multiprocessing`, который предлагает дополнительные и переносимые механизмы IPC, являющиеся частью его универсального API запуска процессов, включая разделяемую память, а также каналы и очереди для передачи произвольных объектов Python в сериализованном виде. Но сначала познакомимся с более традиционными подходами.

Анонимные каналы

Каналы, как механизм взаимодействия программ, реализуются операционной системой, а стандартная библиотека Python лишь обеспечивает доступ к ним. Каналы – это однонаправленные потоки ввода-вывода, по своему действию напоминающие буфер в совместно используемой памяти, интерфейс которого с обеих сторон похож на простой файл. В самом типичном случае использования 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 для работы с файлами.

Большую часть работы с каналами проделывает операционная система. Например, функции для чтения данных из канала обычно *блокируют* вызывающую программу, пока данные не станут доступны (то есть будут отправлены программой на другом конце), вместо того чтобы возвращать признак конца файла. Кроме того, функция чтения из канала всегда возвращает самые «старые» данные, записанные в канал, то есть каналы реализуют модель «первым пришел, первым ушел» – данные, которые были записаны раньше, будут прочитаны в первую очередь. Такие особенности позволяют использовать каналы для синхронизации выполнения независимых программ.

Каналы бывают двух видов – *анонимные* и *именованные*. Именованные каналы (иногда их называют «fifo») представляются в компьютере в виде файла. Так как именованные каналы фактически являются внешними файлами, взаимодействующие процессы вообще могут быть не связаны родственными узами – они могут быть совершенно независимыми программами.

Анонимные каналы, напротив, существуют только внутри процессов и обычно используются вместе с приемом *ветвления* процессов, как средство связи родительского и порожденного дочернего процессов в приложении: родитель и потомок общаются через совместно используемые дескрипторы файлов каналов. Потоки выполнения действуют в одном и том же процессе и совместно используют всю глобальную память, поэтому анонимные каналы могут использоваться и для взаимодействий между ними.

Основы анонимных каналов

Поскольку анонимные каналы являются наиболее традиционным инструментом, мы познакомимся с ними в первую очередь. Сценарий в примере 5.19 создает копию вызывающего процесса с помощью функции `os.fork` (с ветвлением процессов мы познакомились выше в этой главе). После ветвления исходный родительский процесс и его дочерняя копия общаются между собой через канал, созданный функцией `os.pipe` перед ветвлением. Функция `os.pipe` возвращает кортеж с двумя *дескрипторами файлов* – низкоуровневыми идентификаторами файлов, с которыми мы познакомились в главе 4, представляющими входной и выходной концы канала. Так как ответвленный дочерний процесс получает *копии* дескрипторов файлов своего родителя, то при записи в дескриптор выходного конца канала в дочернем процессе данные посылаются обратно родителю по каналу, созданному до создания дочернего процесса.

Пример 5.19. PP4E\System\Processes\pipe1.py

```
import os, time

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)                    # заставить родителя подождать
        msg = ('Spam %03d' % zzz).encode() # каналы - двоичные файлы
        os.write(pipeout, msg)             # отправить данные 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:                     # создать копию процесса
        child(pipeout)                     # в копии вызвать child
    else:                                  # в родителе слушать канал
        while True:
            line = os.read(pipein, 32)     # остановиться до получения данных
```

```
            print('Parent %d got [%s] at %s' % (os.getpid(), line,
                                                time.time()))

parent()
```

Если запустить эту программу в Linux, Cygwin или в другой Unix-подобной системе (функция `pipe` имеется в стандартной реализации Python для Windows, а вот функция `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 инициации обмена.[1] Для имитации длительных операций дочерний процесс заставляет родителя ждать каждое следующее сообщение на одну секунду дольше предыдущего с помощью вызова функции `time.sleep`, пока задержка не достигнет четырех секунд. Когда счетчик задержки `zzz` становится равным 005, он сбрасывается обратно в 000, и отсчет начинается сначала:

```
[C:\...\PP4E\System\Processes]$ python pipe1.py
Parent 6716 got [b'Spam 000'] at 1267996104.53
Parent 6716 got [b'Spam 001'] at 1267996105.54
Parent 6716 got [b'Spam 002'] at 1267996107.55
Parent 6716 got [b'Spam 003'] at 1267996110.56
Parent 6716 got [b'Spam 004'] at 1267996114.57
Parent 6716 got [b'Spam 000'] at 1267996114.57
Parent 6716 got [b'Spam 001'] at 1267996115.59
Parent 6716 got [b'Spam 002'] at 1267996117.6
Parent 6716 got [b'Spam 003'] at 1267996120.61
Parent 6716 got [b'Spam 004'] at 1267996124.62
Parent 6716 got [b'Spam 000'] at 1267996124.62
Parent 6716 got [b'Spam 001'] at 1267996125.63
```

...и так далее: Ctrl-C для выхода...

---

[1] Понятия «клиент» и «сервер» будут разъяснены в части книги, посвященной приложениям для Интернета. Там взаимодействие будет организовано с помощью сокетов (с которыми мы встретимся далее в этой главе и которые грубо можно назвать двунаправленными каналами связи между программами, выполняющимися на компьютерах, соединенных сетью, или на одном и том же компьютере), 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 (выполнять ветвление при этом не требуется). Но как мы увидим, модели сокетов повсеместно используются большинством протоколов сценариев для Интернета – сообщения электронной почты, например, являются, по сути, всего лишь форматированными строками, передаваемыми между программами через сокеты со стандартными номерами портов, зарезервированными для протоколов обмена электронной почтой.

Обратите внимание, что родитель принимает из канала строку байтов. Данные через простые каналы обычно передаются в виде строк байтов, если они обслуживаются с применением инструментов для работы с дескрипторами файлов, с которыми мы встречались в главе 4 (как мы видели там, инструменты чтения из дескрипторов и записи в дескрипторы, имеющиеся в модуле `os`, всегда возвращают и принимают строки байтов). Именно поэтому мы вынуждены в дочернем процессе вручную кодировать текст в строку байтов перед записью в канал – операция форматирования строк не может применяться к строкам байтов. Как будет показано в следующем разделе, дескриптор канала можно обернуть объектом текстового файла, как мы делали это в примерах главы 4, но этот прием обеспечит лишь автоматическое кодирование и декодирование при передаче данных средствами объекта, тогда как внутри канала данные все равно будут передаваться в форме строк байтов.

Обертывание дескрипторов канала объектами файлов

При внимательном рассмотрении вывода предыдущего сценария можно заметить, что когда счетчик задержки в дочернем процессе достигает значения 004, родительский процесс получает из канала сразу два сообщения – дочерний процесс записывает два различных сообщения, но на некоторых платформах или при определенных настройках (отличных от тех, что используются здесь) они могут оказаться достаточно близки по времени и получены родителем как один блок данных. В действительности родитель каждый раз слепо запрашивает чтение не более 32 байтов, но получает тот текст, который есть в канале.

Чтобы отделять одно сообщение от другого, можно определить для канала символ-разделитель. Для этого можно использовать символ конца строки, так как можно обернуть дескриптор канала объектом файла с помощью функции `os.fdopen` и использовать его метод `readline` для поиска в канале очередного разделителя `\n`. Кроме того, этот прием позволит использовать более мощные инструменты объектов текстовых файлов, с которыми мы познакомились в главе 4. Такая схема реализована в примере 5.20.

Пример 5.20. PP4E\System\Processes\pipe2.py

```
# аналогичен сценарию pipe1.py, но обертывает входной дескриптор канала
# объектом файла для обеспечения построчного чтения данных,
# и в обоих процессах закрывает неиспользуемый дескриптор канала

import os, time

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)                      # заставить родителя подождать
        msg = ('Spam %03d\n' % zzz).encode() # каналы - двоичные файлы в 3.X
```

```
        os.write(pipeout, msg)                # отправить данные родителю
        zzz = (zzz+1) % 5                     # переход к 0 через 5 итераций

def parent():
    pipein, pipeout = os.pipe()               # создать канал с 2 концами
    if os.fork() == 0:                        # дочерний процесс пишет в канал
        os.close(pipein)                      # закрыть дескриптор ввода
        child(pipeout)
    else:                                     # в родителе слушать канал
        os.close(pipeout)                     # закрыть дескриптор вывода
        pipein = os.fdopen(pipein)            # создать объект текстового файла
        while True:
            line = pipein.readline()[:-1]     # остановиться до получения данных
            print('Parent %d got [%s] at %s' % (os.getpid(), line,
                                                time.time()))

parent()
```

Эта версия расширена тем, что *закрывает* неиспользуемые концы каналов в каждом процессе (например, после ветвления родительский процесс закрывает свою копию выходного конца канала, в который пишет дочерний процесс); обычно программы должны закрывать неиспользуемые концы каналов. В этой новой версии родителю при каждом чтении из канала гарантированно возвращается одно сообщение дочернего процесса, потому что при записи все они разделяются маркерами:

```
[C:\...\PP4E\System\Processes]$ python pipe2.py
Parent 8204 got [Spam 000] at 1267997789.33
Parent 8204 got [Spam 001] at 1267997790.03
Parent 8204 got [Spam 002] at 1267997792.05
Parent 8204 got [Spam 003] at 1267997795.06
Parent 8204 got [Spam 004] at 1267997799.07
Parent 8204 got [Spam 000] at 1267997799.07
Parent 8204 got [Spam 001] at 1267997800.08
Parent 8204 got [Spam 002] at 1267997802.09
Parent 8204 got [Spam 003] at 1267997805.1
Parent 8204 got [Spam 004] at 1267997809.11
Parent 8204 got [Spam 000] at 1267997809.11
Parent 8204 got [Spam 001] at 1267997810.13
```

...и так далее: Ctrl-C для выхода...

Обратите внимание, что в этой версии текстовые данные теперь возвращаются в виде объекта `str`, так как функция `os.fdopen` по умолчанию устанавливает режим `r` при открытии файла. Как уже упоминалось, обмен данными через каналы обычно происходит с использованием строк байтов, когда дескрипторы используются непосредственно, с применением инструментов из модуля `os`, но обертывание дескрипторов объектами файлов позволяет использовать для представления данных строки `str`. В этом примере декодирование байтов в строку `str` в родительском процессе выполняется операцией чтения. Использо-

вание функции os.fdopen и текстового режима в дочернем процессе позволило бы избежать необходимости кодирования данных вручную, но это кодирование в любом случае выполнялось бы объектом файла (хотя кодирование символов ASCII, как в данном примере, является достаточно тривиальной операцией). Что касается простых файлов, лучший режим обработки данных в канале определяется самой их природой.

Анонимные каналы и потоки выполнения

Функция os.fork, используемая в примерах из предыдущего раздела, недоступна в стандартной версии Python для Windows, но функция os.pipe доступна. Так как все потоки выполнения работают в рамках одного процесса и совместно используют дескрипторы файлов (и всю глобальную память), это позволяет использовать анонимные каналы для синхронизации потоков выполнения. Это, возможно, более низкоуровневый механизм, чем очереди или общие объекты, и тем не менее он обеспечивает дополнительное средство организации взаимодействий между потоками выполнения. Так, в примере 5.21 демонстрируется тот же способ обмена данными с помощью канала, но уже между потоками, а не между процессами.

Пример 5.21. PP4E\System\Processes\pipe-thread.py

```
# анонимные каналы и потоки выполнения вместо процессов;
# эта версия работает и в Windows

import os, time, threading

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)                    # заставить родителя подождать
        msg = ('Spam %03d' % zzz).encode() # каналы - двоичные файлы
        os.write(pipeout, msg)             # отправить данные родителю
        zzz = (zzz+1) % 5                  # переход к 0 после 4

def parent(pipein):
    while True:
        line = os.read(pipein, 32)         # остановиться до получения данных
        print('Parent %d got [%s] at %s' % (os.getpid(), line, time.time()))

pipein, pipeout = os.pipe()
threading.Thread(target=child, args=(pipeout,)).start()
parent(pipein)
```

Так как стандартная версия Python для Windows поддерживает потоки выполнения, данный сценарий будет работать и в Windows. Вывод сценария похож на предыдущий, но взаимодействующими сторонами здесь являются потоки выполнения, а не процессы (обратите внимание, что из-за бесконечных циклов по крайней мере один из потоков

выполнения может не завершиться после нажатия комбинации Ctrl-C – чтобы остановить процесс *python.exe*, выполняющий этот сценарий, в Windows может потребоваться вызвать Диспетчер задач (Task Manager) или закрыть окно консоли):

```
C:\...\PP4E\System\Processes> pipe-thread.py
Parent 8876 got [b'Spam 000'] at 1268579215.71
Parent 8876 got [b'Spam 001'] at 1268579216.73
Parent 8876 got [b'Spam 002'] at 1268579218.74
Parent 8876 got [b'Spam 003'] at 1268579221.75
Parent 8876 got [b'Spam 004'] at 1268579225.76
Parent 8876 got [b'Spam 000'] at 1268579225.76
Parent 8876 got [b'Spam 001'] at 1268579226.77
Parent 8876 got [b'Spam 002'] at 1268579228.79
```

...и так далее: Ctrl-C или Диспетчер задач для выхода...

Двунаправленный обмен данными с помощью анонимных каналов

Обычно каналы позволяют данным перемещаться только в одном направлении – один конец является входом, другой выходом. А как быть, если потребуется организовать общение между программами в обоих направлениях? Например, одна программа может посылать другой запрос на информацию и ждать получения этой информации. Один канал не может справиться с такими двунаправленными переговорами, но это можно реализовать с помощью двух каналов: один канал используется для передачи запроса, а второй – для пересылки ответа запросившей программе.

Так происходит во множестве практических применений. Например, однажды я написал графический интерфейс для отладчика командной строки C-подобного языка программирования и связал два процесса каналами, используя описываемый прием. Графический интерфейс запускался как отдельный процесс, который конструировал и отправлял команды отладчику командной строки в его поток ввода через канал, а затем анализировал результаты, возвращаемые отладчиком через его поток вывода. Графический интерфейс выступал в роли программиста, вводящего команды с клавиатуры, и как клиент отладчика-сервера. Вообще говоря, возможность запускать программы командной строки как дочерние процессы, с потоками ввода-вывода, подключенными к каналам, позволяет добавлять новые интерфейсы к старым программам. Простой пример реализации графического интерфейса подобного рода мы увидим в главе 10.

Модуль в примере 5.22 демонстрирует один из способов реализации идеи связывания стандартных потоков ввода и вывода двух программ. В нем функция `spawn` запускает новую дочернюю программу и соединяет потоки ввода и вывода родительской программы с потоками ввода и вывода дочерней программы. Это означает, что:

- **Когда родитель читает из своего стандартного потока ввода, происходит чтение текста, отправленного дочерней программой в свой стандартный поток вывода.**
- **Когда родитель записывает в свой стандартный поток вывода, происходит отправка данных в стандартный поток ввода дочерней программы.**

В итоге две независимые программы обмениваются между собой данными через свои стандартные потоки ввода-вывода.

Пример 5.22. PP4E\System\Processes\pipes.py

```
"""
запускает дочерний процесс/программу, соединяет свои потоки stdin/stdout
с потоками stdout/stdin дочернего процесса -- операции чтения и записи на
стороне родительского процесса отображаются на стандартные потоки ввода-вывода
дочерней программы; напоминает соединение потоков с помощью модуля subprocess;
"""

import os, sys

def spawn(prog, *args):           # имя программы, аргументы командной строки
    stdinFd = sys.stdin.fileno()          # получить дескрипторы 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 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 после ветвления:
        os.close(parentStdout)            # закрыть родительские концы
        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)             # запустить новую программу
        assert False, 'execvp failed!'    # os.exec никогда не вернется сюда

if __name__ == '__main__':
    mypid = os.getpid()
    spawn('python', 'pipes-testchild.py', 'spam')  # породить дочернюю прогр.

    print('Hello 1 from parent', mypid)            # в stdin дочерней прогр.
    sys.stdout.flush()                             # вытолкнуть буфер stdio
    reply = input()                                # из потока вывода потомка
    sys.stderr.write('Parent got: "%s"\n' % reply) # stderr не связан
                                                   # с каналом!
```

```
    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 управлением стандартной версии Python для Windows (не забывайте, что функции `fork` в этой системе пока нет). В действительности большинство функций, используемых в этом модуле, отображаются непосредственно в системные вызовы Unix (и могут ужаснуть разработчиков, которые не пишут для Unix!). С некоторыми из этих функций мы уже встречались (например, `os.fork`), но значительная часть этого программного кода основывается на концепциях Unix, разобраться с которыми должным образом в данной книге нам не позволит время. Тем не менее ниже приводится упрощенное описание системных вызовов, использованных в этом примере:

`os.fork`

Создает копию вызывающего процесса и возвращает числовой идентификатор ID дочернего процесса только родительскому процессу.

`os.execvp`

Затирает вызывающий процесс новой программой. Эта функция очень похожа на использовавшуюся выше функцию `os.execlp`, но принимает *кортеж* или *список* аргументов командной строки (в аргументе `*args` в заголовке функции).

`os.pipe`

Возвращает кортеж дескрипторов файлов, представляющих входной и выходной концы канала, как показано в приведенных ранее примерах.

`os.close(fd)`

Закрывает файл с дескриптором `fd`.

`os.dup2(fd1, fd2)`

Копирует всю системную информацию, связанную с файлом, заданным дескриптором `fd1`, в файл, заданный дескриптором `fd2`.

Что касается стандартных потоков ввода-вывода, самое важное место здесь занимает функция `os.dup2`. Например, вызов `os.dup2(parentStdin, stdinFd)` по сути присваивает дескриптор файла `stdin` родительского процесса входному концу одного из создаваемых каналов – все операции чтения из потока `stdin` с этого момента будут извлекать данные из канала. После соединения другого конца этого канала с копией файла потока `stdout` дочернего процесса посредством `os.dup2(childStdout, stdoutFd)` текст, выводимый дочерним процессом в его поток `sdtdout`, будет отправляться через канал в поток `stdin` родителя. По своему эффекту этот прием напоминает способ, которым мы соединяли потоки ввода-вывода с помощью модуля `subprocess` в главе 3, но этот сценарий менее переносим и действует на более низком уровне.

Для проверки этой утилиты в конце файла помещен программный код самотестирования, который запускает в дочернем процессе программу, приведенную в примере 5.23, и производит операции чтения и записи в стандартные потоки ввода-вывода, осуществляя обмен данными через два канала.

Пример 5.23. PP4E\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)      # приостановить родительский процесс
    recv = input()     # stdin связан с каналом: данные будут поступать из
                       # родительского потока вывода stdout
    time.sleep(3)
    send = 'Child %d got: [%s]' % (mypid, recv)
    print(send)        # stdout связан с каналом: данные будут поступать в
                       # родительский поток ввода stdin
    sys.stdout.flush() # гарантировать отправку, иначе процесс заблокируется
```

Ниже приводятся результаты тестирования в Cygwin (напоминает Unix-подобные системы, такие как Linux). Вывод не производит большого впечатления, но показывает, как две программы выполняются независимо и обмениваются данными через каналы, управляемые операционной системой. Этот пример еще более напоминает модель клиент/сервер (если представить себе дочерний процесс как сервер, отвечающий на запросы родителя). Текст, заключенный в квадратные скобки, попал из родительского процесса в дочерний и вернулся обратно в родительский, и все это через каналы, подключенные к стандартным потокам ввода-вывода:

```
[C:\...\PP4E\System\Processes]$ python pipes.py
Child 9228 of 9096 got arg: "spam"
Parent got: "Child 9228 got: [Hello 1 from parent 9096]"
Parent got: "Child 9228 got: [Hello 2 from parent 9096]"
```

Еще раз о буферизации потока вывода: взаимоблокировки и выталкивание буферов

Два процесса из примера в предыдущем разделе ведут простой диалог, но этого вполне достаточно, чтобы проиллюстрировать некоторые опасности, таящиеся в процедурах обмена данными между программами. Во-первых, отметим, что обе программы должны выводить сообщения в поток `stderr` – их потоки `stdout` подключены к потокам ввода другой программы. Поскольку процессы используют общие дескрипторы файлов, получается, что в родительском и в дочернем процессе `stderr` – это

один и тот же поток, поэтому сообщения будут выводиться 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 функцию `sys.stdout.flush`. Запрос ввода из канала обычно блокирует вызывающий процесс, если в канале нет данных, но в нашем примере из-за этого не должно возникать проблем, потому что запись производится столько же раз, сколько чтение на другом конце канала. Однако по умолчанию поток `sys.stdout` *буферизуется*, поэтому выведенный текст в действительности может оказаться переданным только через некоторое время (когда до конца будут заполнены буферы вывода). На практике, если принудительно не выталкивать содержимое 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 *взаимоблокировки* (deadlock), когда оба процесса блокируются в вызове функции `input` и ожидают события, которое никогда не произойдет.

С технической точки зрения, для потока вывода `stdout` по умолчанию используется режим *построчной буферизации*, когда он подключен к терминалу, а когда он подключается к другим устройствам, таким как файлы, сокеты или каналы, для него используется режим *полной буферизации*. Это объясняет, почему текст при выводе в окно консоли появляется на экране немедленно, а не когда процесс завершит работу или когда буфер вывода окажется заполнен, как в случаях, когда поток вывода подключен к какому-то другому устройству.

Буферизация выходных данных в действительности производится системными библиотеками, используемыми для доступа к каналам, а не самими каналами (каналы помещают выходные данные в очередь, но не скрывают их от чтения!). На самом деле в данном примере буферизация выполняется только потому, что мы передаем информацию для канала через `sys.stdout` – встроенный объект файла, по умолчанию выполняющий буферизацию. Однако такие аномалии могут происходить и при использовании других инструментов взаимодействия процессов.

В целом, когда программы ведут такого рода двусторонний диалог, избежать взаимоблокировки, связанной с буферизацией, можно несколькими способами:

- *Выталкивание буферов*: Как показано в примерах 5.22 и 5.23, выталкивание выходных буферов потоков вывода в канал с помощью метода `flush` объекта файла является простым способом принудительной очистки буферов. Для выталкивания выходного буфера потока вывода, используемого функцией `print`, используйте метод `sys.stdout.flush`.
- *Аргументы*: Как говорилось выше в этой главе, если вызывать интерпретатор Python с ключом `-u` командной строки, он отключит полную буферизацию потока вывода `sys.stdout` в выполняемых им программах. Запись любого непустого значения в переменную окру-

жения PYTHONUNBUFFERED эквивалентна передаче этого ключа в команду запуска всех программ.

- *Режимы открытия*: Имеется также возможность использовать каналы в небуферизованном режиме. Для этого можно использовать низкоуровневые функции из модуля os для чтения и записи в дескрипторы канала или передавать в аргументе функции os.fdopen, определяющем размер буфера, значение 0 (*небуферизованный* режим) или 1 (режим *построчной буферизации*), чтобы отключить буферизацию в объекте файла, обертывающем дескриптор. Для управления режимом буферизации вывода в *файлы fifo* (описываются в следующем разделе) можно также использовать аргументы функции open. Обратите внимание, что в Python 3.X полностью небуферизованный режим возможен только для двоичных файлов и невозможен для текстовых.
- *Каналы команд*: Как упоминалось выше в этой главе, точно так же можно определять аргументы, управляющие буферизацией, для каналов командной строки, когда они создаются функциями os.popen и subprocess.Popen, но они воздействуют на конец канала в вызывающем процессе, и не влияют на режим буферизации в порожденных программах. Следовательно, этот прием не в состоянии предотвратить задержку вывода из последних, но может использоваться для передачи текстовых данных в каналы ввода других программ.
- *Сокеты*: Как мы увидим далее, функция socket.makefile принимает похожий аргумент, определяющий режим буферизации для сокетов (описываются далее в этой главе и книге), но в Python 3.X требует обязательную буферизацию для текстовых данных и, похоже, не поддерживает построчный режим буферизации (подробнее об этом в главе 12).
- *Инструменты*: Для решения более сложных задач можно также использовать высокоуровневые инструменты, которые фактически обманывают программу, заставляя ее полагать, что она подключена к терминалу. Эти инструменты предназначены для работы с программами не на языке Python, в которых невозможно организовать выталкивание буферов вручную или использовать ключ -u. Дополнительные подробности приводятся во врезке «Подробнее о буферизации потоков ввода-вывода: pty и Pexpect» ниже.

Использование дополнительных потоков выполнения позволяет избежать блокирования главного потока, управляющего графическим интерфейсом, но в действительности это решение лишь переносит проблему из одного места в другое (дочерний поток точно так же может оказаться заблокированным). Из предложенных решений, перечисленных выше, первые два – выталкивание буферов вручную и аргументы командной строки – часто являются наиболее простыми. Фактически, благодаря удобству в использовании, второй из перечисленных выше приемов заслуживает, чтобы сказать о нем несколько слов. Попробуйте

следующее: закомментируйте все вызовы метода `sys.stdout.flush` в примерах 5.22 и 5.23 (в файлах *pipes.py* и *pipes-testchild.py*) и измените вызов функции, порождающий дочерний процесс в файле *pipes.py*, как показано ниже (то есть добавьте ключ `-u` командной строки):

```
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 выталкивании выходного буфера потока вывода `stdout` вручную, потому что теперь поток вывода `stdout` будет действовать в небуферизованном режиме.

Мы еще будем рассматривать эффекты, связанные с отсутствием буферизации потоков вывода, в главе 10, где напишем простой графический интерфейс, отображающий вывод программы командной строки, который будет приниматься через неблокирующий сокет и через канал в потоке выполнения. Еще раз, более подробно мы исследуем эту тему в главе 12, где будем использовать более универсальные способы перенаправления стандартных потоков ввода-вывода в сокеты. В целом, однако, взаимоблокировка представляет собой более обширную проблему, для полного исследования которой здесь недостаточно места. С другой стороны, если у вас достаточно знаний, чтобы пытаться использовать механизмы IPC в языке Python, то, наверное, вы уже ветеран войн со взаимоблокировками.

Анонимные каналы обеспечивают возможность общения процессов, связанных родственными узами, но они не подходят для программ, запускаемых независимо друг от друга. Чтобы обеспечить общение между такими программами, необходимо перейти к следующему разделу и исследовать механизмы, обладающие более широкой областью видимости.

Подробнее о буферизации потоков ввода-вывода: pty и Pexpect

В Unix-подобных системах для принудительного перевода потока стандартного вывода других программ в небуферизованный режим можно также использовать модуль `pty` из стандартной библиотеки Python, что особенно удобно, если эти другие программы написаны не на языке Python и вы не имеете возможности изменить их программный код.

Технически режим буферизации потока вывода `stdout` в других программах определяется за пределами интерпретатора Python, посредством проверки – подключен ли дескриптор потока вывода к терминалу. Эта проверка выполняется в стандартной библиотеке ввода-вывода файловой системы, и ее результаты не могут контролироваться порождаемыми программами.

В целом, когда стандартный поток вывода подключен к терминалу, для него используется режим построчной буферизации, а при подключении к другим устройствам (включая файлы, каналы и сокеты) используется режим полной буферизации. Такая политика применяется с целью повышения эффективности. Файлы и потоки ввода-вывода, создаваемые внутри сценариев на языке Python, следуют тем же правилам, но в них вы можете явно указать политику буферизации с помощью инструментов создания файлов.

Модуль `pty` фактически обманывает порожденную программу, заставляя ее думать, что она подключена к терминалу, благодаря чему в буфере потока вывода `stdout` сохраняется только одна строка. В результате появление в потоке вывода каждого нового символа перевода строки приводит к выталкиванию предыдущей строки из буфера, что типично для интерактивных программ и именно то, что нужно, если вы предполагаете получать строки по мере их воспроизведения.

Однако обратите внимание, что модуль pty не требуется применять для изменения режима буферизации потоков ввода-вывода при запуске сценариев на языке Python: просто используйте ключ `-u` командной строки, передавайте инструментам создания файлов аргументы, определяющие режим построчной буферизации, или вручную вызывайте метод `sys.stdout.flush()` в порождаемых программах. Кроме того, модуль `pty` на сегодняшний день доступен в Python не на всех платформах (он имеется в версии Python для Cygwin, но отсутствует в стандартной версии для Windows).

Пакет `Pexpect`, эквивалент программы `expect` для Unix на языке Python, использует модуль `pty` для обеспечения дополнительной функциональности и взаимодействий в обход стандартных потоков ввода-вывода (например, для ввода пароля). Более подробную информацию о модуле `pty` вы можете найти в руководстве по библиотеке Python, а также попробуйте поискать информацию о пакете `Pexpect` в Интернете.

Именованные каналы (fifo)

На некоторых платформах имеется возможность создавать каналы, существующие в виде настоящих файлов в файловой системе. Такие файлы называются именованными каналами (named pipes), или «fifo», так как они ведут себя в точности как каналы, которые создавались в программах из предыдущего раздела. Однако, вследствие того, что именованные каналы связаны с настоящими файлами, располагающимися на компьютере и являющимися внешними для любой программы, они никак не связаны с памятью, совместно используемой заданиями, и мо-

гут использоваться, как механизм взаимодействий между потоками, процессами и программами, запускаемыми независимо друг от друга.

После создания файла именованного канала процессы открывают его по имени и осуществляют чтение и запись в него с использованием обычных файловых операций. Fifo являются однонаправленными потоками. В типичной ситуации серверная программа читает данные из fifo, а одна или более клиентских программ записывают в него данные. Кроме того, для реализации двусторонней связи можно использовать группу из двух fifo, точно так же, как это делалось в предыдущем разделе с использованием анонимных каналов.

Так как именованные каналы являются файлами, они живут дольше, чем анонимные каналы внутри процессов, и к ним могут обращаться программы, запускаемые независимо. Приводившиеся выше примеры использования неименованных каналов основывались на том факте, что дескрипторы файлов (в том числе каналов) копируются в память дочерних процессов. Это осложняет использование анонимных каналов для организации взаимодействий программ, запускаемых независимо. С помощью же fifo доступ к каналам производится по имени файла, которое видят все программы независимо от наличия отношений родитель-потомок между процессами. Фактически, подобно обычным файлам, fifo обычно живут дольше программ, использующих их. Однако, в отличие от обычных файлов, операционная система синхронизирует доступ к fifo, что делает их идеальным механизмом IPC.

Благодаря этим отличиям именованные каналы лучше подходят в качестве универсального механизма IPC для независимых программ, взаимодействующих по схеме клиент/сервер. Например, постоянно выполняющаяся программа сервера может создавать каналы fifo и ждать из них запросы, поступающие от произвольных клиентов, а не только от тех, что могли бы быть порождены сервером. В некотором смысле, именованные каналы составляют альтернативу сокетам, с которыми мы встретимся в следующем разделе. Однако, в отличие от сокетов, при использовании каналов fifo нет прямой возможности устанавливать сетевые соединения с удаленными компьютерами, они не поддерживаются в версии Python для Windows на сегодняшний день, и для доступа к ним используется стандартный интерфейс для работы с файлами вместо более уникальных номеров портов и функций, которые мы будем изучать далее.

Основы именованных каналов

В Python файлы именованных каналов создаются с помощью функции `os.mkfifo`, которая доступна в настоящее время только в Unix-подобных системах и в версии Python для Cygwin в Windows, но недоступна в стандартной версии Python для Windows. Эта функция просто создает внешний файл – для отправки и получения данных через fifo его нужно открывать и обрабатывать, как стандартный файл.

Для иллюстрации в примере 5.24 приводится измененная версия сценария *pipe2.py* из примера 5.20, в которой вместо анонимных каналов используются именованные каналы. Как и сценарий *pipe2.py*, эта версия открывает в дочернем процессе канал fifo с помощью функции os.open в режиме двоичного доступа, а в родительском процессе – с помощью встроенной функции open, в текстовом режиме. Вообще говоря, на любом конце канала можно использовать любой из предложенных приемов, чтобы интерпретировать данные в канале как двоичные данные или как текст.

Пример 5.24. PP4E\System\Processes\pipefifo.py

```
"""
именованные каналы; функция os.mkfifo недоступна в Windows (без Cygwin);
здесь нет необходимости использовать прием ветвления процессов, потому что
файлы каналов fifo являются внешними по отношению к процессам -- совместное
использование дескрипторов файлов в родителе/потомке здесь неактуально;
"""

import os, time, sys
fifoname = '/tmp/pipefifo'                    # имена должны быть одинаковыми

def child():
    pipeout = os.open(fifoname, os.O_WRONLY) # открыть fifo как дескриптор
    zzz = 0
    while True:
        time.sleep(zzz)
        msg = ('Spam %03d\n' % zzz).encode() # был открыт в двоичном режиме
        os.write(pipeout, msg)
        zzz = (zzz+1) % 5

def parent():
    pipein = open(fifoname, 'r')              # открыть fifo как текстовый файл
    while True:
        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 именованный канал
    if len(sys.argv) == 1:
        parent()      # если нет аргументов - запустить как родительский процесс
    else:             # иначе - как дочерний процесс
        child()
```

Поскольку канал fifo существует независимо от родительского и дочернего процессов, нет никакой необходимости использовать прием ветвления процессов: дочерний процесс может быть запущен независимо от родительского и должен лишь открыть файл fifo с таким же именем. Ниже, например, в Cygwin, родитель запущен в одном окне командной

строки, а потомок – в другом. Сообщения начинают появляться в окне родителя только после того, как потомок будет запущен и начнет записывать сообщения в файл fifo:

```
[C:\...\PP4E\System\Processes] $ python pipefifo.py        # окно родителя
Parent 8324 got "Spam 000" at 1268003696.07
Parent 8324 got "Spam 001" at 1268003697.06
Parent 8324 got "Spam 002" at 1268003699.07
Parent 8324 got "Spam 003" at 1268003702.08
Parent 8324 got "Spam 004" at 1268003706.09
Parent 8324 got "Spam 000" at 1268003706.09
Parent 8324 got "Spam 001" at 1268003707.11
Parent 8324 got "Spam 002" at 1268003709.12
Parent 8324 got "Spam 003" at 1268003712.13
Parent 8324 got "Spam 004" at 1268003716.14
Parent 8324 got "Spam 000" at 1268003716.14
Parent 8324 got "Spam 001" at 1268003717.15
```

...и так далее: Ctrl-C для выхода...

```
[C:\...\PP4E\System\Processes]$ file /tmp/pipefifo         # окно потомка
/tmp/pipefifo: fifo (named pipe)

[C:\...\PP4E\System\Processes]$ python pipefifo.py -child
```

...Ctrl-C для выхода...

Области применения именованных каналов

Благодаря отображению точек взаимодействий в файловую систему, доступную всем программам, выполняющимся на компьютере, именованные каналы способны решать самые разнообразные задачи взаимодействий между процессами на платформах, где они поддерживаются. Например, хотя сценарий, представленный в этом разделе, запускает независимые программы, именованные каналы могут также использоваться как механизм взаимодействий между потоками выполнения внутри процесса и между процессами, связанными отношением родитель/потомок, практически так же, как анонимные каналы.

Однако, благодаря поддержке взаимодействий независимых программ, файлы fifo могут найти более широкое применение в моделях взаимодействий клиент/сервер. Например, с помощью именованных каналов можно сделать связь отладчика командной строки с графическим интерфейсом, о реализации которой на основе анонимных каналов я рассказывал выше, более гибкой – при использовании файлов fifo для соединения потоков ввода-вывода графического интерфейса с потоками ввода-вывода отладчика командной строки, графический интерфейс можно было бы запускать независимо.

Подобную функциональность предоставляют сокеты, которые к тому же подкупают свойственной им возможностью передачи данных по

сети и переносимостью на платформу Windows, о чем рассказывается в следующем разделе.

Сокеты: первый взгляд

Сокеты, реализация которых на языке Python находится в модуле `socket`, представляют собой более универсальный механизм IPC, чем каналы, которые мы рассматривали перед этим. Сокеты позволяют передавать данные не только между программами, выполняющимися на одном и том же компьютере, но и между программами, выполняющимися на разных компьютерах, соединенных сетью. При использовании сокетов для реализации механизма взаимодействий между процессами, выполняющимися на одном и том же компьютере, программы подключаются к сокетам, используя глобальный для этого компьютера номер порта, и передают данные. При использовании сокетов для выполнения сетевых соединений программы указывают сетевое имя компьютера и номер порта и обмениваются данными с удаленными программами.

Основы сокетов

Сокеты – это один из наиболее часто используемых инструментов IPC, тем не менее невозможно до конца понять их API, не понимая его роль в сетевых взаимодействиях. Вследствие этого я отложу подробное освещение особенностей сокетов, пока мы не исследуем порядок их использования в сетевых приложениях в главе 12. В этом разделе дается краткое введение и предварительный обзор сокетов, благодаря которому вы сможете сравнить их с именованными каналами (fifo), представленными в предыдущем разделе. В двух словах:

- Подобно именованным каналам, сокеты являются глобальным для компьютера механизмом – они не требуют наличия памяти, совместно используемой потоками выполнения или процессами, и поэтому могут использоваться независимыми программами.
- В отличие от именованных каналов, сокеты идентифицируются по номеру порта, а не по имени файла в файловой системе, – при работе с ними используется совершенно иной API, не похожий на файлы, тем не менее имеется возможность обертывать их объектами файлов. Сокеты обладают более высокой степенью переносимости: они поддерживаются практически на всех платформах, включая стандартную версию Python для Windows.

Кроме того, сокеты могут играть роли, выходящие далеко за рамки взаимодействий между процессами и за рамки этой главы. Тем не менее, чтобы проиллюстрировать основные особенности использования сокетов, в примере 5.25 приводится сценарий, который запускает сервер и 5 клиентов в виде потоков выполнения, работающих параллельно на одном компьютере и обменивающихся данными через сокеты. Так как

все клиенты подключаются к одному и тому же порту, сервер получает данные, отправляемые всеми клиентами.

Пример 5.25. PP4E\System\Processes\socket_preview.py

```
"""
использует сокеты для обмена данными между заданиями: запускает потоки
выполнения, взаимодействующие с помощью сокетов; независимые программы также
могут использовать сокеты для взаимодействий, потому что они принадлежат
системе в целом, как и именованные каналы; смотрите части книги, посвященные
разработке графических интерфейсов и сценариев для Интернета, где приводятся
более практичные примеры использования сокетов; некоторым серверам может
потребоваться взаимодействовать через сокеты с клиентами в виде потоков
выполнения и процессов; данные через сокеты передаются в виде строк байтов, но
точно так же через них можно передавать сериализованные объекты или кодированный
текст Юникода;
ВНИМАНИЕ: при обращении к функции print в потоках выполнения может потребоваться
синхронизировать их, если есть вероятность перекрытия по времени;
"""

from socket import socket, AF_INET, SOCK_STREAM  # переносимый API сокетов

port = 50008           # номер порта, идентифицирующий сокет
host = 'localhost'     # сервер и клиент выполняются на локальном компьютере

def server():
    sock = socket(AF_INET, SOCK_STREAM)   # IP-адрес TCP-соединения
    sock.bind(('', port))                 # подключить к порту на этой машине
    sock.listen(5)                        # до 5 ожидающих клиентов
    while True:
        conn, addr = sock.accept()        # ждать соединения с клиентом
        data = conn.recv(1024)            # прочитать байты данных от клиента
        reply = 'server got: [%s]' % data # conn - новый подключенный сокет
        conn.send(reply.encode())         # отправить байты данных клиенту

def client(name):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.connect((host, port))            # подключить сокет к порту
    sock.send(name.encode())              # отправить байты данных серверу
    reply = sock.recv(1024)               # принять байты данных от сервера
    sock.close()                          # до 1024 байтов в сообщении
    print('client got: [%s]' % reply)

if __name__ == '__main__':
    from threading import Thread
    sthread = Thread(target=server)
    sthread.daemon = True                 # не ждать завершения потока сервера
    sthread.start()                       # ждать завершения дочерних потоков
    for i in range(5):
        Thread(target=client, args=('client%s' % i,)).start()
```

Внимательно рассмотрите программный код этого примера и комментарии, чтобы получить представление о том, как используются методы объекта сокета для передачи данных. В двух словах, метод accept сокетов данного типа, который вызывается сервером, принимает соединения от клиентов, по умолчанию блокируя выполнение сервера, пока клиент не пришлет запрос на обслуживание, и возвращает новый сокет, соединенный с клиентом. После установления соединения клиент и сервер приступают к обмену строками байтов с помощью методов приема и передачи, вместо записи и чтения. Однако, как будет показано далее в этой книге, сокеты могут обертываться объектами файлов, как это делалось выше с дескрипторами каналов. Кроме того, подобно дескрипторам каналов, необернутые сокеты работают со строками bytes, а не с текстовыми строками str. Это объясняет, почему результат форматирования строк в примере кодируется вручную.

Ниже приводится вывод этого сценария после запуска в Windows:

```
C:\...\PP4E\System\Processes> socket_preview.py
client got: [b"server got: [b'client1']"]
client got: [b"server got: [b'client3']"]
client got: [b"server got: [b'client4']"]
client got: [b"server got: [b'client2']"]
client got: [b"server got: [b'client0']"]
```

В этих результатах нет ничего особенного; каждая строка отражает данные, отправленные клиентом серверу и затем отправленные обратно: сервер принимает строку байтов от клиента и отправляет их обратно, добавив некоторый текст. Так как все потоки выполняются параллельно, клиенты на этом компьютере обслуживаются в случайном порядке.

Сокеты и независимые программы

Потоки выполнения могут использовать сокеты для взаимодействий между собой, однако модель потоков с общей памятью часто позволяет использовать более простые механизмы, такие как глобальные переменные и объекты и очереди. Наиболее ярко сокеты проявляют себя, когда они используются для организации взаимодействий отдельных процессов и программ, запускаемых независимо друг от друга. В примере 5.26 повторно используются функции server и client из предыдущего примера, но теперь они вызываются процессами и потоками из программ, запускаемых независимо друг от друга.

Пример 5.26. PP4E\System\Processes\socket-preview-progs.py

```
"""
тоже сокет, но теперь для общения независимых программ, а не только потоков
выполнения; сервер в этом примере обслуживает клиентов, выполняющихся в виде
отдельных процессов и потоков; сокеты, как и именованные каналы, являются
глобальными для компьютера: для их использования не требуется совместно
используемая память
"""
```

```
from socket_preview import server, client  # оба используют тот же номер порта
import sys, os
from threading import Thread

mode = int(sys.argv[1])
if mode == 1:                             # запустить сервер в этом процессе
    server()
elif mode == 2:                           # запустить клиента в этом процессе
    client('client:process=%s' % os.getpid())
else:                                     # запустить 5 потоков-клиентов
    for i in range(5):
        Thread(target=client, args=('client:thread=%s' % i,)).start()
```

Запустим этот сценарий в Windows (переносимость – важное преимущество сокетов). Сначала запустим в отдельном окне сервер, как независимую программу, – этот процесс будет выполняться без остановки, ожидая от клиентов запросов на соединение (как и в предыдущем примере с каналами, вам может потребоваться воспользоваться Диспетчером задач (Task Manager) или закрыть окно, чтобы прервать работу сервера):

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 1
```

Теперь, в другом окне, запустим несколько клиентов, выполняющихся в виде процессов и потоков, как независимые программы; если передать сценарию аргумент 2 в командной строке, он запустит один клиентский процесс, а если передать 3, он породит пять потоков выполнения, обменивающихся данными с сервером параллельно:

```
C:\...\PP4E\System\Processes> socket-preview-progs.py 2
client got: [b"server got: [b'client:process=7384']"]

C:\...\PP4E\System\Processes> socket-preview-progs.py 2
client got: [b"server got: [b'client:process=7604']"]

C:\...\PP4E\System\Processes> socket-preview-progs.py 3
client got: [b"server got: [b'client:thread=1']"]
client got: [b"server got: [b'client:thread=2']"]
client got: [b"server got: [b'client:thread=0']"]
client got: [b"server got: [b'client:thread=3']"]
client got: [b"server got: [b'client:thread=4']"]

C:\..\PP4E\System\Processes> socket-preview-progs.py 3
client got: [b"server got: [b'client:thread=3']"]
client got: [b"server got: [b'client:thread=1']"]
client got: [b"server got: [b'client:thread=2']"]
client got: [b"server got: [b'client:thread=4']"]
client got: [b"server got: [b'client:thread=0']"]

C:\...\PP4E\System\Processes> socket-preview-progs.py 2
client got: [b"server got: [b'client:process=6428']"]
```

Области применения сокетов

Примеры в этом разделе иллюстрируют основную роль сокетов как механизма IPC, но они не дают полного представления о практической ценности сокетов. Несмотря на то, что они могут работать лишь со строками байтов, совсем не трудно вообразить области возможного их применения. Приложив небольшие усилия, например:

- Через сокеты можно передавать произвольные *объекты* Python, такие как списки и словари (или, по крайней мере, их копии), сериализуя их в строки байтов с помощью модуля `pickle`, который был представлен в главе 1 и подробно рассматривается в главе 17.
- Как мы увидим в главе 10, стандартный поток вывода сценария можно *перенаправить* в окно графического интерфейса, подключив поток вывода сценария к его сокету, который задействован в графическом интерфейсе в неблокирующем режиме.
- Программы, загружающие произвольный *текст* из Интернета, могут читать его в виде строки байтов через сокеты и декодировать вручную, используя имена кодировок, встроенные в заголовки `content-type` или непосредственно в теги самих данных.
- *Интернет в целом* можно рассматривать, как область применения сокетов – как мы увидим в главе 12, в самом конце, электронная почта, FTP и веб-страницы – это просто форматированные строки байтов, доставляемые через сокеты.

Плюс любые другие ситуации, когда необходимо организовать обмен данными между программами, – сокеты являются универсальным, переносимым и гибким инструментом. Например, они могут играть ту же роль, что и каналы fifo в примере с графическим интерфейсом для отладчика, приведенным выше, но при этом сокеты могут использоваться в Windows и позволили бы даже подключать графический интерфейс к отладчику, выполняющемуся на другом компьютере. В силу этого они оцениваются многими как более мощный инструмент IPC.

Повторю еще раз, что вы должны рассматривать этот раздел лишь как краткий обзор. Более полный охват сокетов подразумевает знакомство с сетевыми концепциями, поэтому мы отложим более детальное изучение API сокетов до главы 12. Кроме того, мы встретимся с сокетами в главе 10, где будем исследовать возможность перенаправления потоков ввода-вывода графического интерфейса, о которой говорилось выше, и познакомимся с различными дополнительными областями их применения в части книги, посвященной программированию для Интернета. В четвертой части, например, мы будем использовать сокеты для передачи целых файлов и создадим более надежные серверы, порождающие потоки выполнения или процессы для обмена данными с клиентами, чтобы избежать ситуации отказа в обслуживании. А те-

перь, продолжая тему этой главы, перейдем к еще одному, последнему инструменту IPC, – к сигналам.

Сигналы

Сигналы, в отсутствие лучшей аналогии, можно сравнить с палкой, которой тыкают в процесс. Программы генерируют сигналы, чтобы запустить обработчик данного сигнала в другом процессе. Операционная система тоже этим занимается – некоторые сигналы, генерируемые при необычных системных событиях, могут завершить программу, если их не обработать. Если это несколько напоминает возбуждение исключений в Python, то это неслучайно: сигналы являются программно генерируемыми событиями и аналогом исключений, действующими между процессами. Однако, в отличие от исключений, сигналы идентифицируются по номеру, не помещаются в очередь и в действительности являются механизмом асинхронных событий за пределами интерпретатора Python, управляемым операционной системой.

Чтобы сигналы можно было использовать в сценариях, в состав стандартной библиотеки Python входит модуль `signal`, который позволяет программам на языке Python регистрировать функции в качестве обработчиков сигналов. Этот модуль доступен как в Unix-подобных системах, так и в Windows (хотя в версии для Windows число сигналов может оказаться меньше). Для иллюстрации базового интерфейса сигналов в примере 5.27 приводится сценарий, устанавливающий функцию обработчика сигнала, номер которого передается как аргумент командной строки.

Пример 5.27. PP4E\System\Processes\signal1.py

```
"""
обработка сигналов в Python; номер сигнала N передается как аргумент командной
строки; чтобы передать сигнал этому процессу, используйте команду оболочки "kill
-N pid"; большинство обработчиков сигналов восстанавливаются интерпретатором
после обработки сигнала (смотрите главу, посвященную сетевым сценариям, где
приводится описание сигнала SIGCHLD); в Windows модуль signal также доступен,
но он определяет небольшое количество типов сигналов, а кроме того, в Windows
отсутствует функция os.kill;
"""

import sys, signal, time
def now(): return time.ctime(time.time())    # строка с текущим временем

def onSignal(signum, stackframe):            # обработчик сигнала
    print('Got signal', signum, 'at', now()) # большинство обработчиков
                                             # остаются действующими
signum = int(sys.argv[1])
signal.signal(signum, onSignal)              # установить обработчик сигнала
while True: signal.pause()                   # ждать сигнала (или: pass)
```

Здесь используются только две функции из модуля `signal`:

`signal.signal`

Принимает номер сигнала, объект функции и устанавливает эту функцию в качестве обработчика сигнала с данным номером. Интерпретатор Python автоматически восстанавливает большинство обработчиков сигналов, когда они возникают, поэтому нет необходимости повторно вызывать эту функцию внутри самого обработчика сигнала, чтобы заново его зарегистрировать. То есть обработчики всех сигналов, за исключением сигнала `SIGCHLD`, остаются установленными, пока не будут сброшены явно (например, путем установки его в значение `SIG_DFL`, чтобы восстановить режим по умолчанию, или в значение `SIG_IGN`, чтобы игнорировать сигнал). Поведение сигнала `SIGCHLD` зависит от платформы.

`signal.pause`

Приостанавливает процесс, пока не будет перехвачен следующий сигнал. Функция `time.sleep` действует аналогично, но не работает с сигналами в моей системе Linux, – она генерирует ошибку прерванного системного вызова. Цикл `while True: pass` тоже остановит сценарий, но будет напрасно тратить ресурсы процессора.

Ниже приводится вывод сценария, выполняющегося под управлением Cygwin в Windows (точно так же он будет действовать в любой Unix-подобной системе, такой как Linux): номер ожидаемого сигнала (12) передается в командной строке, а программа запускается в фоновом режиме с помощью оператора оболочки `&` (доступного в большинстве Unix-подобных оболочек):

```
[C:\...\PP4E\System\Processes]$ python signal1.py 12 &
[1] 8224

$ ps
      PID    PPID    PGID     WINPID  TTY  UID    STIME  COMMAND
I    8944       1    8944       8944  con 1004 18:09:54  /usr/bin/bash
     8224    7336    8224      10020  con 1004 18:26:47  /usr/local/bin/python
     8380    7336    8380        428  con 1004 18:26:50  /usr/bin/ps

$ kill -12 8224
Got signal 12 at Sun Mar  7 18:27:28 2010

$ kill -12 8224
Got signal 12 at Sun Mar  7 18:27:30 2010

$ kill -9 8224
[1]+ Killed                  python signal1.py 12
```

Ввод и вывод здесь несколько перемешаны, потому что процесс осуществляет вывод на тот же экран, в котором вводятся новые команды оболочки. Послать программе сигнал можно с помощью команды обо-

лочки kill, которая принимает номер сигнала и ID процесса (8224). Всякий раз, когда очередная команда kill посылает сигнал, процесс отвечает сообщением, сгенерированным функцией обработчика сигнала. Сигнал с номером 9 всегда завершает процесс.

Модуль signal экспортирует также функцию signal.alarm, с помощью которой определяется интервал времени в секундах, по истечении которого должен быть отправлен сигнал SIGALRM. Чтобы определить максимальное время ожидания и обработать ситуацию его превышения, достаточно установить обработчик сигнала SIGALRM и вызвать функцию signal.alarm, как показано в примере 5.28.

Пример 5.28. PP4E\System\Processes\signal2.py

```
"""
установка сигналов по истечении времени ожидания и их обработка на языке Python;
функция time.sleep некорректно ведет себя при появлении сигнала SIGALARM
(как и любого другого сигнала на моем компьютере, работающем под управлением
Linux), поэтому здесь вызывается функция signal.pause, которая приостанавливает
выполнение сценария до получения сигнала;
"""

import sys, signal, time
def now(): return time.asctime()

def onSignal(signum, stackframe):             # обработчик сигнала
    print('Got alarm', signum, 'at', now())   # большинство обработчиков
                                              # остаются действующими
while True:
    print('Setting at', now())
    signal.signal(signal.SIGALRM, onSignal)   # установить обработчик сигнала
    signal.alarm(5)                           # послать сигнал через 5 секунд
    signal.pause()                            # ждать сигнала
```

После запуска этого сценария под управлением Cygwin в Windows функция обработчика onSignal будет вызываться каждые пять секунд:

```
[C:\...\PP4E\System\Processes]$ python signal2.py
Setting at Sun Mar 7 18:37:10 2010
Got alarm 14 at Sun Mar 7 18:37:15 2010
Setting at Sun Mar 7 18:37:15 2010
Got alarm 14 at Sun Mar 7 18:37:20 2010
Setting at Sun Mar 7 18:37:20 2010
Got alarm 14 at Sun Mar 7 18:37:25 2010
Setting at Sun Mar 7 18:37:25 2010
Got alarm 14 at Sun Mar 7 18:37:30 2010
Setting at Sun Mar 7 18:37:30 2010
```

...Ctrl-C для выхода...

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 ввода-вывода 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 используют в комбинации с другими инструментами IPC. Например, начальный сигнал может сообщить программе, что клиент хочет установить связь через именованный канал, – примерно как если 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. Такой способ интеграции иногда может служить альтернативой обращения к блокируемой функции ввода в дочернем потоке выполнения.

Обратите также внимание на функцию `os.kill(`*`pid, sig`*`)`, которая передает сигналы известным процессам из сценариев на языке Python в Unix-подобных системах, – она очень похожа на команду `kill` оболочки, использованную нами выше. Необходимый идентификатор ID можно получить из значения, возвращаемого функцией `os.fork`, порождающей дочерний процесс, или с помощью других функций. Как и `os.fork`, эта функция доступна в версии Cygwin Python, но она отсутствует в стандартной версии Python для Windows. Кроме того, смотрите обсуждение приема использования сигналов для удаления процессов «зомби» в главе 12.

Пакет multiprocessing

Теперь, когда мы познакомились с альтернативными механизмами IPC и получили возможность исследовать процессы, потоки выполнения и обсудили вопросы переносимости и ограничения GIL, накладываемые на потоки выполнения, пришло время познакомиться с еще одной альтернативой, которая стремится предоставить все лучшее из обоих миров. Как упоминалось выше, пакет `multiprocessing` из стандартной библиотеки Python позволяет сценариям порождать процессы, используя API, близко напоминающий модуль `threading`.

Этот относительно новый пакет может использоваться и в Unix, и в Windows, в отличие от низкоуровневого приема ветвления процессов. Он поддерживает платформонезависимую модель запуска процессов и предоставляет сопутствующие инструменты, такие как средства IPC, включая блокировки, каналы и очереди. Кроме того, для выполнения параллельных операций он использует не потоки выполнения, а процессы, что позволяет эффективно обойти ограничения GIL. Из этого следует, что пакет `multiprocessing` позволяет программисту использо-

вать преимущества многопроцессорных систем для выполнения параллельных задач, сохраняя при этом простоту и переносимость модели потоков выполнения.

Зачем нужен пакет multiprocessing?

Так зачем же нам изучать еще одну парадигму параллельной обработки и еще один инструмент, когда у нас уже имеются потоки выполнения, процессы, инструменты IPC, такие как сокеты, каналы и очереди, изучавшиеся выше? Прежде чем погружаться в детали, мне хотелось бы сказать несколько слов о том, зачем вам может (или не может) потребоваться этот пакет. Несмотря на то, что по своей производительности этот пакет не может конкурировать с низкоуровневыми механизмами потоков выполнения и порождения процессов, во многих случаях он может оказаться весьма привлекательным решением:

- В отличие от приема ветвления процессов, этот пакет обеспечивает высокую степень переносимости и предоставляет мощные инструменты IPC.
- В отличие от механизма потоков выполнения, этот пакет обеспечивает по-настоящему параллельное выполнение задач в многопроцессорных системах, хотя и за счет некоторой потери времени, необходимого на выполнение процедуры запуска.

С другой стороны, этот пакет накладывает некоторые ограничения, отсутствующие в механизме потоков выполнения:

- Так как объекты приходится копировать через границы процессов, они не могут располагаться в совместно используемой памяти, как в случае потоков выполнения, – изменения в одном процессе не могут быть замечены в другом процессе. На практике возможность совместного использования памяти может оказаться самой веской причиной использования потоков выполнения, а отсутствие такой возможности в этом пакете может ограничить круг его применения в определенных контекстах.
- Так как этот пакет требует, чтобы процессы в Windows, а также некоторые инструменты IPC поддерживали возможность сериализации, это может усложнить или сделать непереносимой реализацию некоторых парадигм программирования, особенно когда в них предусматривается использование связанных методов или передача несериализуемых объектов, таких как сокеты, в порожденные процессы.

Например, типичный прием использования `lambda`-функций, который прекрасно работает при использовании модуля `threading`, не может использоваться для передачи вызываемых объектов процессам в Windows, потому что они не могут быть сериализованы. Аналогично из-за невозможности сериализовать связанные методы объектов в многопоточных программах может потребоваться применять обходные решения, если

потоки выполняют связанные методы или завершение потоков реализовано как передача им вызываемых объектов (возможно, даже связанных методов) через общие очереди. Модель потоков, выполняющихся в пределах одного и того же процесса, поддерживает возможность непосредственного использования `lambda`-функций и связанных методов, но модель отдельных процессов, реализованная в пакете `multiprocessing`, такой возможности не поддерживает.

В главе 10 мы напишем механизм управления потоками для графических интерфейсов, который опирается на возможность передачи вызываемых объектов через очередь в реализации операций завершения потоков, – вызываемые объекты помещаются в очередь рабочими потоками выполнения, извлекаются и переправляются дальше главным потоком. Многопоточная программа PyMailGUI, которая будет реализована в главе 14, использует этот механизм управления потоками для передачи через очередь связанных методов, реализующих операции завершения потоков, и выполнения этих методов в главном потоке. Эту схему невозможно непосредственно перенести на модель отдельных процессов, реализованную в пакете `multiprocessing`.

Не углубляясь в детали, отмечу: чтобы задействовать пакет `multiprocessing` в приложении PyMailGUI, операции в нем пришлось бы реализовать в виде простых функций или в виде подклассов процессов, чтобы обеспечить возможность сериализации. Хуже того, эти операции может потребоваться реализовать в виде простых идентификаторов, передаваемых через главный процесс, если они обновляют графический интерфейс или изменяют состояние объекта в целом, – сериализация приводит к созданию копии объекта в принимающем процессе и не является ссылкой на оригинал, а операция ветвления в Unix вообще копирует процесс целиком. Модификация изменяемого объекта, полученного из сериализованной копии в новом процессе, например, не окажет никакого влияния на оригинал.

Требование поддержки сериализации для аргументов процессов в Windows может ограничить область применения пакета `multiprocessing` и в других контекстах. Например, в главе 12 мы увидим, что этот пакет не может напрямую использоваться для решения проблемы непереносимости функции `os.fork` при традиционном подходе к разработке *сетевых серверов* в Windows, потому что подключенные сокеты некорректно сериализуются при передаче новому процессу, созданному этим пакетом, и не могут использоваться для общения с клиентом. В этом контексте потоки выполнения обеспечивают более переносимое и, пожалуй, более эффективное решение.

Ситуация с приложениями, которые обмениваются простыми сообщениями, обстоит намного лучше. Ограничения, касающиеся сообщений, проще преодолеть, особенно если эти сообщения составляют часть архитектуры приложения, изначально основанной на применении процессов. Кроме того, в этом пакете имеются другие инструменты, такие

как менеджеры и API разделяемой памяти, которые, хотя и являются узкоспециализированными и не такими универсальными, как память, совместно используемая потоками выполнения, тем не менее способны в некоторых случаях предоставить дополнительные возможности обмена информацией между процессами.

Однако в целом, благодаря тому что пакет `multiprocessing` опирается на использование отдельных процессов, он лучше подходит для реализации относительно независимых задач, которым не требуется общая память и для нормальной работы вполне достаточно возможности обмена сообщениями и инструментов доступа к разделяемой памяти, имеющихся в этом пакете. К их числу можно отнести множество приложений, но это не означает, что с помощью данного пакета можно напрямую заменить любые многопоточные программы, и он не во всех случаях может быть альтернативой приему, основанному на ветвлении процессов.

Чтобы по-настоящему оценить преимущества и слабые стороны этого пакета, обратимся к первому примеру и попутно исследуем реализацию пакета.

Основы: процессы и блокировки

В этой книге не так много места, чтобы можно было дать полную оценку этому сложному пакету, поэтому за более подробным описанием обращайтесь к руководству по библиотеке Python. Но если говорить кратко: большинство интерфейсов в этом пакете отражают интерфейсы модулей `threading` и `queue`, с которыми мы уже встречались, поэтому они должны показаться вам знакомыми. Например, класс `Process` в пакете `multiprocessing` имитирует класс `Thread`, встречавшийся нам выше, из модуля `threading` – он позволяет запускать функции параллельно вызывающему сценарию, только в данном случае функция запускается в отдельном процессе, а не в потоке выполнения. Эти основы иллюстрируются в примере 5.29:

Пример 5.29. PP4E\System\Processes\multi1.py

```
"""
основы применения пакета multiprocessing: класс Process по своему действию
напоминает класс threading.Thread, но выполняет функцию в отдельном процессе,
а не в потоке; для синхронизации можно использовать блокировки, например, для
вывода текста; запускает новый процесс интерпретатора в Windows, порождает
дочерний процесс в Unix;
"""

import os
from multiprocessing import Process, Lock

def whoami(label, lock):
```

```
    msg = '%s: name:%s, pid:%s'
    with lock:
        print(msg % (label, __name__, os.getpid()))

if __name__ == '__main__':
    lock = Lock()
    whoami('function call', lock)

    p = Process(target=whoami, args=('spawned child', lock))
    p.start()
    p.join()

    for i in range(5):
        Process(target=whoami, args=(('run process %s' % i), lock)).start()

    with lock:
        print('Main process exit.')
```

Если запустить этот сценарий, он сначала вызовет функцию непосредственно в процессе; затем запустит эту функцию в новом процессе и дождется его завершения; и наконец, в цикле породит пять параллельно выполняющихся процессов вызовов функции – во всех случаях используется API, идентичный классу `threading.Thread`, который мы изучали выше в этой главе. Ниже приводится результат запуска сценария в Windows. Обратите внимание, что пять дочерних процессов, порождаемых в конце сценария, завершаются уже после своего родителя, что вполне обычное явление для процессов:

```
C:\...\PP4E\System\Processes> multi1.py
function call: name:__main__, pid:8752
spawned child: name:__main__, pid:9268
Main process exit.
run process 3: name:__main__, pid:9296
run process 1: name:__main__, pid:8792
run process 4: name:__main__, pid:2224
run process 2: name:__main__, pid:8716
run process 0: name:__main__, pid:6936
```

Так же как класс `threading.Thread`, встречавшийся нам выше, объект `multiprocessing.Process` может принимать функцию в аргументе `target` с параметрами (как сделано в этом примере) или использоваться в качестве родительского класса для переопределения его метода `run`. Метод `start` вызывает метод `run` в новом процессе, а метод `run` по умолчанию просто вызывает функцию, переданную в аргументе `target`. Кроме того, как и в модуле `threading`, метод `join` ожидает завершения дочернего процесса, а объект `Lock` является одним из инструментов синхронизации процессов – здесь он используется, чтобы избежать смешивания текста, выводимого процессами на платформах, на которых это может происходить (в Windows такого не происходит).

Реализация и правила использования

Технически, с целью обеспечить переносимость этот модуль на разных платформах использует разные инструменты:

- В Unix он использует прием ветвления процессов и вызывает метод `run` объекта `Process` в новом дочернем процессе.
- В Windows он запускает новый процесс интерпретатора, используя инструменты Windows создания процессов, передает сериализованный объект `Process` новому процессу через канал и выполняет команду «python -c» в новом процессе, которая запускает специальную функцию на языке Python в этом пакете. Эта функция читает сериализованную версию объекта `Process`, распаковывает ее и вызывает метод `run`.

Мы уже встречались с сериализацией в главе 1 и будем еще изучать ее далее в книге. На самом деле реализация немного сложнее, чем описано выше, и, конечно же, со временем может изменяться, но это действительно очень интересный трюк. Несмотря на то, что переносимый API в целом скрывает подробности реализации от вашего программного кода, тем не менее существуют некоторые тонкие особенности его использования. Например:

- В Windows логику главного процесса вообще следует вкладывать в условную инструкцию проверки условия `__name__ == __main__`, как это сделано здесь, чтобы модуль можно было импортировать без побочных эффектов. Как мы узнаем в главе 17, при десериализации классов и функций необходимо импортировать вмещающие их модули, что является основным требованием.
- Кроме того, значения глобальных переменных в дочерних процессах на платформе Windows могут отличаться от значений этих же переменных в родительском процессе, имевших место на момент вызова метода `start`, потому что вмещающие их модули будут импортироваться в новом процессе.
- Дополнительно, в Windows все аргументы конструктора `Process` должны быть сериализуемыми объектами. Поскольку к числу этих аргументов относится и аргумент `target`, в нем допускается передавать только простые функции, которые могут быть сериализованы, – в этом аргументе нельзя передавать связанные или несвязанные *методы* объектов и функции, созданные с помощью инструкции `lambda`. Подробнее о правилах сериализации рассказывается в описании модуля `pickle` в руководстве по библиотеке Python – сериализовать можно практически любой объект, но чтобы сериализовать вызываемые объекты, такие как функции и классы, они должны быть доступными для импортирования – эти объекты сериализуются по имени и позднее импортируются для воссоздания байт-кода. В Windows объекты, хранящие системную информацию, такие как подключенные сокеты, вообще не могут использоваться в виде аргу-

ментов конструктора Process, потому что они не могут быть сериализованы.

- Точно так же сериализуемыми объектами должны быть экземпляры подклассов класса Process в Windows. Это относится и к значениям их атрибутов. Объекты, доступные в этом пакете (например, Lock в примере 5.29), поддерживают возможность сериализации и поэтому могут использоваться в виде аргументов конструктора Process и в виде атрибутов подклассов.
- Объекты IPC в этом пакете, с которыми мы встретимся в последующих примерах, такие как Pipe и Queue, принимают только сериализуемые объекты, из-за особенностей их реализации (подробнее об этом рассказывается в следующем разделе).
- В Unix дочерний процесс может использовать глобальные элементы, созданные родительским процессом, однако лучше передавать такие объекты дочернему процессу в виде аргументов конструктора Process, что обеспечит совместимость с Windows и позволит избежать возможных проблем на тот случай, если эти объекты будут утилизированы сборщиком мусора в родительском процессе.

В руководстве по библиотеке приводятся и другие правила. Однако в целом, если вы будете придерживаться правил передачи процессам совместно используемых объектов и использовать инструменты взаимодействий, предоставляемые этим пакетом, ваш программный код будет переносим. Теперь рассмотрим практическое применение некоторых из этих инструментов.

Инструменты IPC: каналы, разделяемая память и очереди

Процессы, создаваемые этим пакетом, всегда могут взаимодействовать с помощью общесистемных инструментов, таких как сокеты и файлы fifo, с которыми мы встречались выше, тем не менее пакет multiprocessing также предоставляет свои переносимые инструменты обмена сообщениями, специально предусмотренные для организации взаимодействий между процессами, порождаемыми этим пакетом:

- Объект Pipe реализует анонимный канал, соединяющий два процесса. При вызове конструктора Pipe он возвращает два объекта Connection, представляющие концы канала. По умолчанию каналы являются двунаправленными и позволяют передавать и принимать любые объекты Python, поддерживающие возможность сериализации. В настоящее время в Unix они используют либо пару соединенных друг с другом сокетов, либо порождаются функцией os.pipe, с которой мы встречались выше, а в Windows они реализованы на основе именованных каналов, специфических для этой платформы. Однако, как и объект Process, описанный выше, переносимый API объекта Pipe скрывает все эти тонкости от вызывающей программы.

- Объекты `Value` и `Array` реализуют общую память процессов/потоков, используемую для обмена данными между процессами. Конструкторы этих объектов возвращают одиночный объект и массив объектов, созданные с помощью модуля `ctypes` в разделяемой памяти, доступ к которой синхронизируется по умолчанию.
- Объект `Queue` играет роль списка FIFO объектов Python, допускающий возможность работы с множеством производителей и потребителей. По сути, эта очередь является каналом, снабженным механизмом блокировки для координации попыток доступа к ней, и наследует ограничения объекта `Pipe`, связанные с требованием к поддержке сериализации объектами, помещаемыми в очередь.

Все эти механизмы обеспечивают возможность безопасной работы с нескольким процессами, поэтому они часто играют роль точек синхронизации взаимодействий и позволяют отказаться от использования низкоуровневых инструментов, таких как блокировки, работая сходным с очередями в потоках, с которыми мы встречались выше, образом. Как обычно, канал (или их пара) может использоваться для реализации модели запрос/ответ. Очереди поддерживают более гибкие модели взаимодействий – фактически для преодоления ограничений GIL, вместо потоков выполнения в графическом интерфейсе можно было бы использовать объекты `Process` и `Queue` из пакета `multiprocessing`, и порождать с их помощью долгоживущие процессы, обменивающиеся результатами. Как упоминалось выше, хотя при таком подходе на некоторых платформах на запуск процессов тратится дополнительное время – в сравнении с потоками выполнения, зато он обеспечивает по-настоящему параллельное выполнение задач, если это позволяет сама платформа.

Важное замечание: каналы (и, соответственно, очереди), реализованные в этом пакете, *выполняют сериализацию* объектов, передаваемых через них, благодаря чему они могут быть реконструированы в принимающем процессе (как мы уже видели, в Windows принимающий процесс может выполняться под управлением полностью независимой копии интерпретатора Python). По этой причине они не поддерживают объекты, которые нельзя сериализовать. Как отмечалось выше, к ним относятся некоторые вызываемые объекты, такие как связанные методы и `lambda`-функции (программный код, нарушающий эти ограничения, смотрите в файле *multi-badq.py* в пакете с примерами для этой книги). Передача объектов с системной информацией также может терпеть неудачу. Большинство остальных типов объектов Python, включая классы и простые функции, прекрасно передаются через каналы и очереди.

Кроме того, имейте в виду, что из-за необходимости сериализации объекты, передаваемые с помощью этих инструментов, в принимающем процессе фактически являются *копиями* – прямые изменения в таких объектах не видимы передавшему их процессу. Это вполне объяснимо,

если вспомнить, что данный пакет запускает независимые процессы с их собственными областями памяти – информация не может так же свободно передаваться между ними, как в потоках выполнения, независимо от используемого инструмента IPC.

Каналы в пакете multiprocessing

Чтобы продемонстрировать приемы работы с инструментами IPC, перечисленными выше, далее приводятся три примера, в которых взаимодействия между родительским и дочерним процессами реализованы тремя разными способами. В примере 5.30 используется простой объект канала, по которому передаются данные между родительским и дочерним процессами.

Пример 5.30. PP4E\System\Processes\multi2.py

```
"""
Реализует взаимодействие с помощью анонимных каналов из пакета multiprocessing.
Возвращаемые 2 объекта Connection представляют концы канала: объекты передаются
в один конец и принимаются из другого конца, хотя каналы по умолчанию являются
двунаправленными
"""

import os
from multiprocessing import Process, Pipe

def sender(pipe):
    """
    передает объект родителю через анонимный канал
    """
    pipe.send(['spam'] + [42, 'eggs'])
    pipe.close()

def talker(pipe):
    """
    передает и принимает объекты из канала
    """
    pipe.send(dict(name='Bob', spam=42))
    reply = pipe.recv()
    print('talker got:', reply)

if __name__ == '__main__':
    (parentEnd, childEnd) = Pipe()
    Process(target=sender, args=(childEnd,)).start() # породить потомка
                                                     # с каналом
    print('parent got:', parentEnd.recv())         # принять от потомка
    parentEnd.close()                              # или может быть закрыт
                                                   # автоматически сборщиком
                                                   # мусора
    (parentEnd, childEnd) = Pipe()
    child = Process(target=talker, args=(childEnd,))
    child.start()
```

```
    print('parent got:', parentEnd.recv())          # принять от потомка
    parentEnd.send({x * 2 for x in 'spam'})         # передать потомку
    child.join()                                    # ждать завершения потомка
    print('parent exit')
```

Ниже приводится вывод этого сценария, запущенного в Windows. Первый потомок просто передает объект родителю, а второй – передает и принимает объекты по одному и тому же каналу:

```
C:\...\PP4E\System\Processes> multi2.py
parent got: ['spam', 42, 'eggs']
parent got: {'name': 'Bob', 'spam': 42}
talker got: {'ss', 'aa', 'pp', 'mm'}
parent exit
```

Объекты каналов в этом модуле делают реализацию взаимодействий между двумя процессами переносимой (и практически тривиальной).

Разделяемая память и глобальные объекты

В примере 5.31 используется разделяемая память, которая служит вводом и выводом порожденных процессов. Чтобы обеспечить переносимость этого приема, необходимо создать объекты с помощью пакета и передать их конструктору `Process`. Последний тест в этом примере («loop 4») представляет, пожалуй, наиболее типичный случай использования разделяемой памяти – распределение заданий по нескольким параллельно выполняющимся процессам.

Пример 5.31. PP4E\System\Processes\multi3.py

```
"""
Реализует взаимодействие с помощью объектов разделяемой памяти из пакета.
В Windows передаваемые объекты используются совместно, а глобальные объекты
- нет. Последняя проверка здесь отражает типичный случай использования:
распределение заданий между процессами.
"""

import os
from multiprocessing import Process, Value, Array

procs = 3               # глобальные переменные, отдельные для каждого процесса,
count = 0               # не являются совместно используемыми

def showdata(label, val, arr):
    """
    выводит значения данных в этом процессе
    """
    msg = '%-12s: pid:%4s, global:%s, value:%s, array:%s'
    print(msg % (label, os.getpid(), count, val.value, list(arr)))

def updater(val, arr):
    """
    обменивается данными через разделяемую память
```

```
    """
    global count
    count += 1               # глобальный счетчик недоступен другим процессам
    val.value += 1           # а передаваемый в объекте - доступен
    for i in range(3): arr[i] += 1

if __name__ == '__main__':
    scalar = Value('i', 0)     # разделяемая память: предусматривает
                               # синхронизацию процессов/потоков
    vector = Array('d', procs) # коды типов из ctypes: int, double

    # вывести начальные значения в родительском процессе
    showdata('parent start', scalar, vector)

    # породить дочерний процесс, передать данные в разделяемой памяти
    p = Process(target=showdata, args=('child ', scalar, vector))
    p.start(); p.join()

    # изменить значения в родителе и передать через разделяемую память,
    # ждать завершения каждого потомка
    # все потомки видят изменения, выполненные в родительском процессе и
    # переданные в виде аргументов (но не в глобальной памяти)

    print('\nloop1 (updates in parent, serial children)...')
    for i in range(procs):
        count += 1
        scalar.value += 1
        vector[i] += 1
        p = Process(target=showdata, args=(('process %s' % i),
                                           scalar, vector))
        p.start(); p.join()

    # то же самое, но потомки запускаются параллельно
    # все они видят результат последней итерации, потому что они хранятся
    # в совместно используемых объектах

    print('\nloop2 (updates in parent, parallel children)...')
    ps = []
    for i in range(procs):
        count += 1
        scalar.value += 1
        vector[i] += 1
        p = Process(target=showdata, args=(('process %s' % i),
                                           scalar, vector))
        p.start()
        ps.append(p)
    for p in ps: p.join()

    # объекты в разделяемой памяти изменяются потомками,
    # ждать завершения каждого из них
```

```
print('\nloop3 (updates in serial children)...')
for i in range(procs):
    p = Process(target=updater, args=(scalar, vector))
    p.start()
    p.join()
    showdata('parent temp', scalar, vector)

# то же самое, но потомки запускаются параллельно

ps = []
print('\nloop4 (updates in parallel children)...')
for i in range(procs):
    p = Process(target=updater, args=(scalar, vector))
    p.start()
    ps.append(p)
for p in ps: p.join()

# глобальная переменная count=6 доступна только родителю
# выведет последние результаты # scalar=12: +6 в родителе, +6 в 6 потомках
showdata('parent end', scalar, vector) # array[i]=8:
                                       # +2 в родителе, +6 в 6 потомках
```

Ниже приводится вывод этого сценария, запущенного в Windows. Проследите, как выполняется этот программный код. Обратите внимание, что глобальная переменная недоступна дочерним процессам для совместного использования в Windows, а переданные им объекты `Value` и `Array` – доступны. Последняя строка в этом выводе отражает изменения, выполненные в разделяемой памяти родительским и дочерними процессами, – все элементы в массиве имеют значение 8.0, потому что все они дважды увеличивались в родительском процессе и по одному разу – в каждом из шести дочерних процессов. Скалярное значение также отражает изменения, выполненные в родительском и в дочерних процессах, но, в отличие от потоков выполнения, в Windows глобальные переменные доступны только вмещающему их процессу:

```
C:\...\PP4E\System\Processes> multi3.py
parent start: pid:6204, global:0, value:0, array:[0.0, 0.0, 0.0]
child       : pid:9660, global:0, value:0, array:[0.0, 0.0, 0.0]

loop1 (updates in parent, serial children)...
process 0   : pid:3900, global:0, value:1, array:[1.0, 0.0, 0.0]
process 1   : pid:5072, global:0, value:2, array:[1.0, 1.0, 0.0]
process 2   : pid:9472, global:0, value:3, array:[1.0, 1.0, 1.0]

loop2 (updates in parent, parallel children)...
process 1   : pid:9468, global:0, value:6, array:[2.0, 2.0, 2.0]
process 2   : pid:9036, global:0, value:6, array:[2.0, 2.0, 2.0]
process 0   : pid:9548, global:0, value:6, array:[2.0, 2.0, 2.0]

loop3 (updates in serial children)...
parent temp : pid:6204, global:6, value:9, array:[5.0, 5.0, 5.0]
```

```
loop4 (updates in parallel children)...
parent end  : pid:6204, global:6, value:12, array:[8.0, 8.0, 8.0]
```

А теперь представьте, что в последнем тесте используется намного больший массив и запускается большее количество дочерних процессов, – тогда вы начнете понимать, какие возможности предоставляет этот пакет для реализации параллельной обработки данных.

Очереди и подклассы

Наконец, помимо простых инструментов запуска и взаимодействия с дочерними процессами, пакет `multiprocessing` дополнительно:

- Позволяет создавать подклассы класса `Process`, обеспечивающего базовую структуру процесса и сохранение информации (так же, как `threading.Thread`, но для процессов).
- Реализует объект `Queue`, который может совместно использоваться любым количеством процессов для удовлетворения более широких потребностей при обмене данными (так же, как `queue.Queue`, но для процессов).

Очереди поддерживают более гибкую модель клиент/сервер. Так, сценарий в примере 5.32 порождает три процесса производителя, отправляющие данные в совместно используемую очередь, и периодически проверяет ее на предмет появления результатов – очень похоже на то, как графический интерфейс собирает результаты параллельных вычислений и выводит их, однако здесь параллельные операции выполняются не потоками, а процессами.

Пример 5.32. PP4E\System\Processes\multi4.py

```
"""
От класса Process можно породить подкласс, так же, как от класса threading.
Thread;
объект Queue действует подобно queue.Queue, но обеспечивает обмен данными между
процессами, а не между потоками выполнения
"""

import os, time, queue
from multiprocessing import Process, Queue # общая очередь для процессов
                                           # очередь - это канал +
                                           # блокировки/семафоры
class Counter(Process):
    label = ' @'
    def __init__(self, start, queue):      # сохраняет данные для
        self.state = start                 # использования в методе run
        self.post = queue
        Process.__init__(self)

    def run(self):                         # вызывается в новом процессе
        for i in range(3):                 # методом start()
            time.sleep(1)
```

```
            self.state += 1
            print(self.label ,self.pid, self.state) # self.pid - pid потомка
            self.post.put([self.pid, self.state])   # stdout совместно
                                                    # используется всеми
    print(self.label, self.pid, '-')

if __name__ == '__main__':
    print('start', os.getpid())
    expected = 9

    post = Queue()
    p = Counter(0, post)   # запустить 3 процесса, использующих общую очередь
    q = Counter(100, post) # потомки являются производителями
    r = Counter(1000, post)
    p.start(); q.start(); r.start()

    while expected:        # родитель потребляет данные из очереди
        time.sleep(0.5)    # очень напоминает графический интерфейс,
        try:               # хотя в ГИ часто используются потоки
            data = post.get(block=False)
        except queue.Empty:
            print('no data...')
        else:
            print('posted:', data)
            expected -= 1

    p.join(); q.join(); r.join()  # дождаться завершения дочерних процессов
    print('finish', os.getpid(), r.exitcode) # exitcode - код завершения
                                             # потомка
```

Обратите внимание, что в этом сценарии:

- Функция `time.sleep` имитирует выполнение длительных операций в процессах-производителях.
- Все четыре процесса совместно используют один и тот же поток вывода – функции `print` выводят текст в одно и то же место, но в Windows их вывод не перемешивается (как мы видели выше, пакет `multiprocessing` предоставляет также совместно используемый объект `Lock`, который при необходимости может использоваться для синхронизации процессов).
- Код завершения дочернего процесса доступен после его завершения в атрибуте `exitcode`.

Если запустить этот сценарий, главный процесс-потребитель будет сообщать о своих попытках извлечения данных из очереди, а дочерние процессы-производители (строки с отступами) будут выводить свои идентификаторы ID процессов и данные.

```
C:\...\PP4E\System\Processes> multi4.py
start 6296
```

```
no data...
no data...
  @ 8008 101
posted: [8008, 101]
  @ 6068 1
  @ 3760 1001
posted: [6068, 1]
  @ 8008 102
posted: [3760, 1001]
  @ 6068 2
  @ 3760 1002
posted: [8008, 102]
  @ 8008 103
  @ 8008 -
posted: [6068, 2]
  @ 6068 3
  @ 6068 -
  @ 3760 1003
  @ 3760 -
posted: [3760, 1002]
posted: [8008, 103]
posted: [6068, 3]
posted: [3760, 1003]
finish 6296 0
```

А теперь представьте, что строки, начинающиеся с символа «@», являются результатами длительных операций, а остальные строки представляют отражение работы главного потока выполнения графического интерфейса; широта возможностей этого пакета наверняка станет для вас более очевидной.

Запуск независимых программ

Как мы узнали выше, независимые программы обычно взаимодействуют между собой с помощью общесистемных инструментов, таких как сокеты и файлы fifo, изученные нами ранее. Процессы, порождаемые с помощью пакета `multiprocessing`, также могут пользоваться этими инструментами, однако благодаря их близкому родству мы можем использовать дополнительные механизмы IPC, предоставляемые этим пакетом.

Как и потоки выполнения, пакет `multiprocessing` предназначен для параллельного выполнения функций, а не для запуска отдельных программ. Порожденные функции могут использовать такие инструменты, как `os.system`, `os.popen` и модуль `subprocess` для запуска других программ, если выполняемая операция может заблокировать вызывающий процесс, но часто нет никакого смысла порождать процесс таким способом, чтобы запустить другую программу (другую программу можно запустить, пропустив этот шаг). Фактически в Windows пакет

multiprocessing в настоящее время использует ту же функцию запуска процессов, что и модуль subprocess, поэтому нет смысла использовать первый из них для запуска двух процессов.

Существует также возможность запускать новые программы из дочерних процессов с помощью инструментов, таких как функции os.exec*, с которыми мы встречались выше. Порождая процесс переносимым способом с помощью пакета multiprocessing и запуская в нем новую программу, мы тем самым запускаем независимую программу и эффективно решаем проблему отсутствия функции os.fork в стандартной версии Python для Windows.

Как правило, запуск новой программы не предполагает передачу каких-либо ресурсов конструктору Process (при запуске новая программа затрет программу, выполнявшуюся до нее в этом процессе), но этот конструктор представляет собой переносимый эквивалент комбинации функций fork/exec в Unix. Кроме того, программы, запущенные таким способом, точно так же могут использовать более традиционные инструменты IPC, такие как сокеты и именованные каналы, с которыми мы встречались выше в этой главе. Этот прием иллюстрируется в примере 5.33.

Пример 5.33. PP4E\System\Processes\multi5.py

```
"""
Использует пакет multiprocessing для запуска независимых программ,
с помощью os.fork или других функций
"""

import os
from multiprocessing import Process

def runprogram(arg):
    os.execlp('python', 'python', 'child.py', str(arg))

if __name__ == '__main__':
    for i in range(5):
        Process(target=runprogram, args=(i,)).start()
    print('parent exit')
```

Этот сценарий запускает 5 экземпляров сценария *child.py* из примера 5.4 в виде независимых процессов и не ждет их завершения. Ниже приводится вывод этого сценария, запущенного в Windows, после удаления лишних строк с системным приглашением к вводу, которые произвольно появляются в середине вывода (в Cygwin сценарий действует точно так же, но в этом случае вывод не перемешивается):

```
C:\...\PP4E\System\Processes> type child.py
import os, sys
print('Hello from child', os.getpid(), sys.argv[1])
```

```
C:\...\PP4E\System\Processes> multi5.py
parent exit
Hello from child 9844 2
Hello from child 8696 4
Hello from child 1840 0
Hello from child 6724 1
Hello from child 9368 3
```

Этот прием невозможно применить к потокам выполнения, потому что все потоки выполняются в пределах одного процесса, – запуск новой программы уничтожит все эти потоки выполнения. Хотя данный прием едва ли будет выполняться с той же скоростью, как комбинация a fork/exec в Unix, он, по крайней мере, предоставляет похожий и переносимый способ для Windows.

И многое другое

Наконец, пакет multiprocessing предоставляет множество других инструментов, не демонстрировавшихся в примерах выше, включая инструменты синхронизации по условиям, событиям и с помощью семафоров, а также локальные и удаленные менеджеры, реализующие серверы для управления совместно используемыми объектами. Так, в примере 5.34 демонстрируется поддержка *пулов* (pools) – групп дочерних процессов, совместно работающих над решением определенной задачи.

Пример 5.34. PP4E\System\Processes\multi6.py

```
"""
Плюс многое другое: пулы процессов, менеджеры, блокировки,
условные переменные,...
"""

import os
from multiprocessing import Pool

def powers(x):
    #print(os.getpid()) # раскомментируйте, чтобы увидеть работу потомков
    return 2 ** x

if __name__ == '__main__':
    workers = Pool(processes=5)

    results = workers.map(powers, [2]*100)
    print(results[:16])
    print(results[-2:])

    results = workers.map(powers, range(100))
    print(results[:16])
    print(results[-2:])
```

После запуска Python равномерно распределит задания между рабочими процессами, выполняющимися параллельно:

```
C:\...\PP4E\System\Processes> multi6.py
[4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
[4, 4]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
[31691265005705735037417580l344, 6338253001141147007483516026881]
```

И немножко меньше...

Справедливости ради следует отметить, что помимо дополнительных инструментов и возможностей пакет `multiprocessing` несет с собой также дополнительные ограничения, кроме тех, что мы уже обсудили (поддержка возможности сериализации, доступ к совместно используемым данным и так далее). Например, взгляните на следующий фрагмент программного кода:

```
def action(arg1, arg2):
    print(arg1, arg2)

if __name__ == '__main__':
    Process(target=action, args=('spam', 'eggs')).start() # оболочка ждет
                                                          # завершения потомка
```

Этот сценарий действует, как ожидается, но если изменить последнюю строку, как показано ниже, он не будет работать в Windows, потому что `lambda`-функции не могут быть сериализованы (в действительности, они не могут быть импортированы):

```
Process(target=(lambda: action('spam', 'eggs'))).start() # не работает! -
                                                         # не сериализуется
```

Это не позволяет использовать распространенный прием программирования с применением lambda-функций для передачи данных в вызовы, который мы часто будем использовать для передачи функций обратного вызова в части книги, посвященной графическим интерфейсам. Кроме того, данная особенность отличает пакет `multiprocessing` от модуля `threading`, который послужил прототипом для этого пакета, – вызовы функций, которые могут использоваться при работе с потоками выполнения, такие как приведены ниже, необходимо преобразовать в вызываемые объекты и аргументы:

```
threading.Thread(target=(lambda: action(2, 4))).start() # но с потоками
                                                        # lambda-функции
                                                        # работают
```

И напротив, некоторые особенности поведения модуля `threading` имитируются в пакете `multiprocessing`, хотите вы этого или нет. Из-за того, что программы, использующие этот пакет, по умолчанию ожидают завершения дочерних процессов, мы вынуждены помечать процессы, уста-

навливая атрибут daemon, если нежелательно, чтобы программный код, как показано ниже, блокировал командную оболочку (технически, родительский процесс пытается завершить дочерние процессы-демоны при выходе, то есть программа может завершиться, только когда все потомки являются демонами, что очень похоже на модуль threading):

```
def action(arg1, arg2):
    print(arg1, arg2)
    time.sleep(5)      # обычно препятствует завершению родителя

if __name__ == '__main__':
    p = Process(target=action, args=('spam', 'eggs'))
    p.daemon = True    # не ждать завершения этого потомка
    p.start()
```

Дополнительные подробности о некоторых из этих проблем вы найдете в руководстве по библиотеке Python. Они являются не непреодолимыми препятствиями, но специальными случаями и потенциальными ловушками. Мы еще вернемся к проблемам lambda-выражений и процессов-демонов в более прагматичном контексте в главе 8, где будем использовать модуль multiprocessing для запуска графических интерфейсов, выполняющихся независимо.

Зачем нужен пакет multiprocessing? Заключение

Как следует из примеров в этом разделе, пакет multiprocessing представляет собой мощную альтернативу, объединяющую в себе переносимость и удобство потоков выполнения со способностью к параллельному выполнению, свойственной процессам, и предлагает дополнительные инструменты IPC для получения кодов возврата и решения других задач, сопутствующих параллельной обработке данных.

Хотелось бы надеяться, что данный раздел позволил вам глубже понять достоинства и недостатки пакета, обсуждавшиеся в начале раздела. В частности, его модель, основанная на выполнении отдельных процессов, препятствует свободному использованию общей памяти, как это делается в потоках выполнения, не позволяет передавать связанные методы и lambda-функции из-за ограничений, связанных с сериализацией, а также из-за особенности реализации механизма запуска процессов в Windows. Кроме того, в Windows он требует, чтобы аргументы процесса были сериализуемыми объектами, что препятствует их использованию в серверах для организации взаимодействий с клиентами.

Хотя он и не может служить универсальной заменой модуля threading, тем не менее во многих ситуациях пакет multiprocessing предлагает достаточно привлекательное решение. В частности, для задач параллельного программирования, когда ограничения пакета легко преодолимы, он способен предложить такие качества, как переносимость и произ-

водительность, которые отсутствуют в других, более низкоуровневых многозадачных инструментах Python.

К сожалению, в данной книге не так много места, чтобы дать более всесторонний охват этого пакета помимо этого краткого введения. За дополнительными подробностями обращайтесь к руководству по библиотеке Python. А теперь мы обратим наше внимание к следующей группе дополнительных инструментов запуска программ, знакомством с которыми закончим эту главу.

Другие способы запуска программ

До сих пор в этой книге мы видели разные способы запуска программ – от комбинации функций `os.fork/exec` в Unix до переносимых способов запуска команд, таких как функции `os.system`, `os.popen`, модуль `subprocess` и переносимые механизмы из пакета `multiprocessing`, представленные в последнем разделе. Тем не менее в стандартной библиотеке Python существуют и другие способы, часть из которых менее зависимы от типа платформы, а часть – менее понятны по сравнению с другими. Данный раздел завершает главу кратким обзором этого набора инструментов.

Семейство функций os.spawn

Функции `os.spawnv` и `os.spawnve` впервые были представлены как инструменты запуска программ в Windows, напоминающие по своему действию комбинацию функций `fork/exec` в Unix-подобных системах. На сегодняшний день эти функции доступны на обеих платформах, в Windows и в Unix-подобных системах, а кроме того, были добавлены варианты, повторяющие функциональность других членов семейства `os.exec`.

В последние версии Python был включен переносимый модуль `subprocess` с целью заменить эти функции. Фактически руководство по библиотеке Python включает примечание, отмечающее, что в составе этого модуля имеются более мощные и эквивалентные инструменты, которым следует отдавать предпочтение перед функциями `os.spawn`. Кроме того, новейший пакет `multiprocessing`, в совокупности с функциями `os.exec`, позволяет добиться тех же результатов переносимым способом, как мы уже видели выше. Тем не менее функции семейства `os.spawn` по-прежнему доступны и могут встретиться вам в действующих сценариях на языке Python.

Функции из семейства `os.spawn` запускают программу, указанную в командной строке, в виде нового процесса в Windows и в Unix-подобных системах. По своему действию они напоминают комбинацию функций `fork/exec` в Unix и могут использоваться как альтернатива функциям `system` и `popen`, с которыми мы уже познакомились. В следующем приме-

ре выполняется запуск программы на языке Python двумя традиционными способами (во втором случае дополнительно выполняется чтение стандартного потока вывода программы):

```
C:\...\PP4E\System\Processes> python
>>> print(open('makewords.py').read())
print('spam')
print('eggs')
print('ham')

>>> import os
>>> os.system('python makewords.py')
spam
eggs
ham
0

>>> result = os.popen('python makewords.py').read()
>>> print(result)
spam
eggs
ham
```

Функции, эквивалентные функции `os.spawn`, имеющие чуть более сложную сигнатуру, что обеспечивает более полный контроль над способом запуска программы, – позволяют получить тот же эффект:

```
>>> os.spawnv(os.P_WAIT, r'C:\Python31\python', ('python', 'makewords.py'))
spam
eggs
ham
0

>>> os.spawnl(os.P_NOWAIT, r'C:\Python31\python', 'python', 'makewords.py')
1820
>>> spam
eggs
ham
```

Из всех этих способов функция `spawn` больше всего напоминает прием ветвления программ в Unix. В действительности она не копирует вызывающий процесс (поэтому операции, использующие общие дескрипторы, не работают), но может использоваться для запуска программы Windows, выполняемой совершенно независимо от вызвавшей программы. Сценарий в примере 5.35 делает сходство с шаблонами программирования в Unix еще более очевидным. Он запускает программу с помощью комбинации `fork/exec` в Unix-подобных системах (включая оболочку Cygwin) или вызывает `os.spawnv` в Windows.

Пример 5.35. PP4E\System\Processes\spawnv.py

```
"""
запускает параллельно 10 копий child.py; для запуска программ в Windows
использует spawnv (как fork+exec); флаг P_OVERLAY обозначает замену, флаг
P_DETACH перенаправляет stdout потомка в никуда; можно также использовать
переносимые инструменты из модуля subprocess или из пакета multiprocessing!
"""

import os, sys

for i in range(10):
    if sys.platform[:3] == 'win':
        pypath = sys.executable
        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.')
```

Чтобы понять, как действуют эти примеры, вам необходимо познакомиться с аргументами, которые передаются функциям `spawn`. В этом сценарии мы передаем функции `os.spawnv` флаг режима запуска процесса, полный путь к выполняемому файлу интерпретатора Python и кортеж строк, представляющих команду оболочки, запускающую новую программу. Путь к выполняемому файлу интерпретатора Python доступен сценариям как `sys.executable`. В общем случае флаг *режима запуска процесса* может состоять из следующих предопределенных значений:

`os.P_NOWAIT` и `os.P_NOWAITO`

Функции `spawn` возвращают управление сразу после запуска нового процесса и возвращают его числовой идентификатор ID. Доступны в Unix и Windows.

`os.P_WAIT`

Функции `spawn` не возвращают управление, пока новый процесс не завершится, и в случае успеха возвращают код его завершения, в противном случае – отрицательный номер сигнала («-signal»), если работа процесса была прервана сигналом. Доступен в Unix и Windows.

`os.P_DETACH` и `os.P_OVERLAY`

Флаг `P_DETACH` похож на флаг `P_NOWAIT`, но при этом новый процесс отсоединяется от консоли вызывающего процесса. Если был использован флаг `P_OVERLAY`, текущая программа будет замещена (как при использовании функции `os.exec`). Доступен в Windows.

Фактически в семействе `spawn` насчитывается восемь различных функций. Все они выполняют запуск программ, но несколько отличаются

сигнатурами. Символ «l» в их именах означает, что аргументы программы передаются в виде списка, символ «p» означает, что поиск выполняемого файла программы будет производиться с учетом системного пути, а символ «e» означает, что функции может быть передан словарь, определяющий окружение порождаемой программы: функция `os.spawnve`, например, действует так же, как функция `os.spawnv`, но принимает в дополнительном четвертом аргументе словарь, определяющий иное окружение для порождаемой программы (по умолчанию порождаемые процессы наследуют окружение родительского процесса):

```
os.spawnl(mode, path, ...)
os.spawnle(mode, path, ..., env)
os.spawnlp(mode, file, ...)           # только в Unix
os.spawnlpe(mode, file, ..., env)     # только в Unix
os.spawnv(mode, path, args)
os.spawnve(mode, path, args, env)
os.spawnvp(mode, file, args)          # только в Unix
os.spawnvpe(mode, file, args, env)    # только в Unix
```

Имена этих функций повторяют имена и сигнатуры функций из семейства `os.exec`, поэтому дополнительные подробности, касающиеся отличий между их вариантами, вы найдете в описании функций `os.exec`, выше в этой главе. В отличие от функций `os.exec` только половина функций `os.spawn`, не использующих системный путь (то есть без символа «p» в их именах), в настоящее время реализованы в версии Python для Windows. В Windows поддерживаются все флаги режимов запуска процессов, но флаги `os.P_DETACH` и `os.P_OVERLAY` недоступны в Unix. Со временем перечисленные особенности могут измениться, поэтому обязательно проверьте их описание в руководстве по библиотеке Python или запустите встроенную функцию `dir`, передав ей имя модуля `os` после его импортирования.

Ниже приводится вывод сценария из примера 5.35, запущенного в Windows. Он порождает 10 копий программы *child.py*, с которой мы встречались выше в этой главе:

```
C:\...\PP4E\System\Processes> type child.py
import os, sys
print('Hello from child', os.getpid(), sys.argv[1])

C:\...\PP4E\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 выводят свою информацию в случайном порядке, а родительская программа завершается раньше, чем завершатся все дочерние; все эти программы действительно выполняются в Windows параллельно. Обратите также внимание, что вывод дочерней программы появляется в окне консоли, где был запущен сценарий *spawnv.py*, – при использовании флага `P_NOWAIT` стандартный вывод попадает на родительскую консоль, но отправляется в никуда, если использовать флаг `P_DETACH` (что не является ошибкой при порождении программ с графическим интерфейсом).

После того как я продемонстрировал эту функцию, следует отметить, что оба модуля, `subprocess` и `multiprocessing`, на сегодняшний день предлагают более переносимые альтернативные способы запуска программ с использованием командной строки. Фактически если функции `os.spawn` не обеспечивают вам какого-то уникального поведения, без которого вы не можете обойтись (например, управление всплывающим окном консоли в Windows), платформозависимые отрезки кода, присутствующие в примере 5.35, можно было бы полностью заменить переносимыми инструментами из пакета `multiprocessing`, использованными в примере 5.33.

Функция os.startfile в Windows

Если на сегодняшний день функции `os.spawn` могут рассматриваться, как излишество, то в пользу других инструментов можно привести достаточно веские аргументы. Например, функция `os.system` может использоваться в Windows для запуска команды `start` DOS, которая открывает (то есть запускает) файлы в соответствии с ассоциациями для расширений имен файлов в Windows, как это делается при выполнении двойного щелчка на файле. Функция `os.startfile`, появившаяся в последних версиях Python, делает эту операцию еще более простой, и, в отличие от некоторых других инструментов, позволяет избежать блокирования вызывающего процесса.

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 команда *`command`* вводилась в диалоговом окне Windows Выполнить (Run), которое можно открыть с помощью меню кнопки Пуск (Start). Если *`command`* является именем файла, он открывается точно так же, как если щелкнуть на его имени в графическом интерфейсе Проводника Windows (Windows Explorer).

Например, следующие три команды DOS автоматически запускают Internet Explorer, программу просмотра файлов графических изображений, и программу проигрывания звуковых файлов для соответствую-

щих файлов в командах. Windows просто открывает файл в той программе, которая определена для обработки файлов с указанным расширением. Более того, все три эти программы выполняются независимо от окна консоли DOS, в котором введена команда:

```
C:\...\PP4E\System\Media> start lp4e-preface-preview.html
C:\...\PP4E\System\Media> start ora-lp4e.jpg
C:\...\PP4E\System\Media> start sousa.au
```

Поскольку команда `start` может запустить любой файл и командную строку, нет причин, по которым ее нельзя было бы использовать для запуска независимо выполняемой программы Python:

```
C:\...\PP4E\System\Processes> start child.py 1
```

Это возможно благодаря тому, что при установке Python регистрируется для открытия файлов с расширениями *.py*. Сценарий *child.py* будет запущен независимо от окна консоли DOS, несмотря на то, что не было задано ни имя выполняемого файла интерпретатора Python, ни путь к нему. Однако, поскольку *child.py* просто выводит сообщение и завершается, результат не вполне удовлетворителен: новое окно DOS появляется, чтобы обслужить стандартный вывод сценария, и тут же исчезает, когда он завершается. Лучше будет, если добавить в конец программы вызов функции `input`, чтобы перед завершением происходило ожидание нажатия какой-либо клавиши:

```
C:\...\PP4E\System\Processes> type child-wait.py
import os, sys
print('Hello from child', os.getpid(), sys.argv[1])
input("Press <Enter>") # предотвращает исчезновение окна консоли в Windows

C:\...\PP4E\System\Processes> start child-wait.py 2
```

Теперь окно DOS дочерней программы появляется на экране и сохраняется после возврата из команды `start`. Нажатие клавиши `Enter` во всплывающем окне DOS заставляет его закрыться.

Использование команды start в сценариях Python

Мы знаем, что функции `os.system` и `os.popen` можно вызывать в сценариях для запуска любых команд, которые можно ввести в командной строке DOS, поэтому из сценариев на языке Python можно запускать независимо выполняемые программы простым выполнением команды DOS `start`. Например:

```
C:\...\PP4E\System\Media> python
>>> import os
>>> cmd = 'start lp4e-preface-preview.html' # запустит броузер IE
>>> os.system(cmd)                          # как независимую программу
0
```

Вызов функции `os.system` в этом примере запустит броузер веб-страниц, зарегистрированный в вашей системе как средство просмотра файлов

с расширением *.html* (если только эти программы уже не выполняются). Запущенные программы выполняются совершенно независимо от сеанса Python – при выполнении команды DOS `start` функция `os.system` не ждет завершения запущенной программы.

Функция os.startfile

Команда `start` оказалась настолько удобной, что в последние версии Python была добавлена функция `os.startfile`, которая по сути выполняет те же действия, что и команда DOS `start`, выполняемая с помощью функции `os.system`, и действует, как если бы на указанном файле был выполнен двойной щелчок. Например, следующие вызовы имеют похожий эффект:

```
>>> os.startfile('lp-code-readme.txt')
>>> os.system('start lp-code-readme.txt')
```

На моем компьютере, работающем под управлением Windows, оба вызова открывают текстовый файл в программе Блокнот (Notepad). Однако, в отличие от второго способа, функция `os.startfile`, не предоставляет возможности задержать закрытие запущенного приложения (что достигается передачей ключа `/WAIT` команде DOS `start`) и не позволяет получить код завершения приложения (возвращаемый функцией `os.system`).

В последних версиях Windows следующий вызов также имеет похожий эффект, потому что в них для выполнения команд задействуется реестр (однако такая форма вызова блокируется до закрытия программы просмотра файлов, как при использовании команды `start /WAIT`):

```
>>> os.system('lp-code-readme.txt') # команду 'start' можно не указывать
```

Это довольно удобный способ открытия произвольных документов и медиафайлов, но имейте в виду, что функция `os.startfile` работает только в Windows, потому что она использует реестр Windows, чтобы определить, как открывать файл. Существуют и другие, еще более запутанные и непереносимые способы запуска программ, включая инструменты в пакете PyWin32, который мы не будем рассматривать здесь. Если вам требуется обеспечить переносимость своих сценариев, используйте инструменты запуска программ, из числа представленных выше, такие как функции `os.popen` или `os.spawnv`. Но еще лучше напишите модуль, скрывающий тонкости за переносимым интерфейсом, как показано в следующем, заключительном разделе.

Переносимый модуль запуска программ

Из-за всех этих различий в запуске программ на разных платформах может оказаться трудным запомнить, какие средства должны использоваться в конкретной ситуации. Более того, некоторые из этих инструментов вызываются способами, которые настолько сложны, что быстро

забываются. В настоящее время существуют модули, такие как subprocess и multiprocessing, предлагающие полностью переносимые механизмы, тем не менее для конкретной платформы порой лучше подходят другие инструменты, обладающие более тонкими особенностями поведения. В Windows, например, часто бывает желательно подавить вывод окна командной оболочки.

Мне настолько часто приходится писать сценарии, которым требуется запускать программы на языке Python, что в итоге я создал модуль, постаравшись скрыть в нем большую часть закулисных деталей. Скрыв детали реализации за переносимым интерфейсом, я получил возможность изменять их и использовать новые инструменты, которые появятся в будущем, не влияя на работоспособность программного кода, использующего этот модуль. Работая над 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 5.36 приводится исходный текст модуля, в котором 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 (при необходимости) определяют метод announce для вывода имени программы при запуске.

Пример 5.36. PP4E\launchmodes.py

```
"""
###############################################################################
запускает программы Python с помощью механизмов командной строки и классов
схем запуска; автоматически вставляет "python" и/или путь к выполняемому файлу
интерпретатора в начало командной строки; некоторые из инструментов в этом
модуле предполагают, что выполняемый файл 'python' находится в системном пути
поиска (смотрите Launcher.py);

можно было бы использовать модуль subprocess, но он сам использует функцию
os.popen(), и к тому же цель этого модуля состоит в том, чтобы запустить
независимую программу, а не подключиться к ее потокам ввода-вывода;
можно было бы также использовать пакет multiprocessing, но данный модуль
предназначен для выполнения программ, а не функций: не имеет смысла запускать
процесс, когда можно использовать одну из уже имеющихся возможностей;

новое в этом издании: при запуске сценария передает путь к файлу сценария
через функцию normpath(), которая в Windows замещает все / на \; исправьте
соответствующие участки программного кода в PyEdit и в других сценариях;
вообще в Windows допускается использовать / в командах открытия файлов, но этот
символ может использоваться не во всех инструментах запуска программ;
###############################################################################
"""
```

```
import sys, os
pyfile = (sys.platform[:3] == 'win' and 'python.exe') or 'python'
pypath = sys.executable     # использовать sys в последних версиях Python

def fixWindowsPath(cmdline):
    """
    замещает все / на \ в путях к сценариям в начале команд;
    используется только классами, которые запускают инструменты, требующие
    этого в Windows; в других системах в этом нет необходимости (например,
    os.system в Unix);
    """
    splitline = cmdline.lstrip().split(' ')      # разбить по пробелам
    fixedpath = os.path.normpath(splitline[0])   # заменить прямые слешы
    return ' '.join([fixedpath] + splitline[1:]) # снова объединить в строку

class LaunchMode:
    """
    при вызове экземпляра класса выводится метка и запускается команда;
    подклассы форматируют строки команд для метода run(), если необходимо;
    команда должна начинаться с имени запускаемого файла сценария Python
    и не должна начинаться со слова "python" или с полного пути к нему;
    """
    def __init__(self, label, command):
        self.what = label
        self.where = command
    def __call__(self):          # вызывается при вызове экземпляра,
        self.announce(self.what) # например как обработчик щелчка на кнопке
        self.run(self.where)     # подклассы должны определять метод run()
    def announce(self, text):    # подклассы могут переопределять метод
        print(text)              # announce() вместо логики if/elif
    def run(self, cmdline):
        assert False, 'run must be defined'

class System(LaunchMode):
    """
    запускает сценарий Python, указанный в команде оболочки
    внимание: может блокировать вызывающую программу,
    если в Unix не добавить &
    """
    def run(self, cmdline):
        cmdline = fixWindowsPath(cmdline)
        os.system('%s %s' % (pypath, cmdline))

class Popen(LaunchMode):
    """
    запускает команду оболочки в новом процессе
    внимание: может блокировать вызывающую программу, потому что
    канал закрывается немедленно
    """
    def run(self, cmdline):
        cmdline = fixWindowsPath(cmdline)
```

```
        os.popen(pypath + ' ' + cmdline)   # предполагается, что нет данных
                                           # для чтения
class Fork(LaunchMode):
    """
    запускает команду в явно созданном новом процессе
    только для Unix-подобных систем, включая cygwin
    """
    def run(self, cmdline):
        assert hasattr(os, 'fork')
        cmdline = cmdline.split()                # превратить строку в список
        if os.fork() == 0:                       # запустить новый процесс
            os.execvp(pypath, [pyfile] + cmdline) # запустить новую программу

class Start(LaunchMode):
    """
    запускает команду, независимую от вызывающего процесса
    только для Windows: использует ассоциации с расширениями имен файлов
    """
    def run(self, cmdline):
        assert sys.platform[:3] == 'win'
        cmdline = fixWindowsPath(cmdline)
        os.startfile(cmdline)

class StartArgs(LaunchMode):
    """
    только для Windows: в аргументах могут присутствовать символы прямого
    слеша
    """
    def run(self, cmdline):
        assert sys.platform[:3] == 'win'
        os.system('start ' + cmdline)  # может создать окно консоли

class Spawn(LaunchMode):
    """
    запускает python в новом процессе, независимом от вызывающего,
    для Windows и Unix; используйте P_NOWAIT для окна dos;
    символы прямого слеша допустимы
    """
    def run(self, cmdline):
        os.spawnv(os.P_DETACH, pypath, (pyfile, cmdline))

class Top_level(LaunchMode):
    """
    запускает тот же процесс в новом окне
    на будущее: требуется информация о классе графического интерфейса
    """
    def run(self, cmdline):
       assert False, 'Sorry - mode not yet implemented'

#
# выбор "лучшего" средства запуска для данной платформы
```

```
# возможно, выбор придется уточнить в других местах
#

if sys.platform[:3] == 'win':
    PortableLauncher = Spawn
else:
    PortableLauncher = Fork

class QuietPortableLauncher(PortableLauncher):
    def announce(self, text):
        pass

def selftest():
    file = 'echo.py'
    input('default mode...')
    launcher = PortableLauncher(file, file)
    launcher()                                  # не блокирует

    input('system mode...')
    System(file, file)()                        # блокирует

    if sys.platform[:3] == 'win':
        input('DOS start mode...')              # не блокирует
        StartArgs(file, file)()

if __name__ == '__main__': selftest()
```

Ближе к концу файла модуль выбирает класс по умолчанию, исходя из значения атрибута `sys.platform`: в Windows в атрибут `PortableLauncher` записывается класс, использующий `spawnv`, и класс, использующий комбинацию `fork/exec`, на других платформах. В последних версиях Python можно было бы использовать функцию `spawnv` на всех платформах, но альтернативные инструменты в этом модуле могут использоваться в других контекстах. 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* – методом `__call__` перегрузки операторов, вместо самого метода; поэтому классы этого модуля можно также использовать для создания обработчиков обратного вызова в графических интерфейсах на базе 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 графического интерфейса другой

программы. Связать инструмент запуска с нажатием кнопки в графическом интерфейсе можно следующим способом:

```
Button(root, text=name, command=PortableLauncher(name, commandLine))
```

При автономном выполнении, как обычно, вызывается функция `self-test` этого модуля. При использовании класса `System` вызывающий процесс блокируется до завершения запускаемой программы, а при использовании `PortableLauncher` (в действительности, `Spawn` или `Fork`) и `Start` – нет.

```
C:\...\PP4E> type echo.py
print('Spam')
input('press Enter')

C:\...\PP4E> python launchmodes.py
default mode...
echo.py
system mode...
echo.py
Spam
press Enter
DOS start mode...
echo.py
```

Практическое применение этого файла мы увидим в главе 8, где он будет использоваться для запуска диалога с графическим интерфейсом, и в нескольких примерах в главе 10, включая PyDemos и PyGadgets, – сценарии, предназначенные для обеспечения переносимого способа запуска основных примеров в этой книге, которые находятся в вершине дерева примеров. Эти сценарии просто импортируют `PortableLauncher` и регистрируют экземпляры, которые будут откликаться на события в графическом интерфейсе, поэтому они прекрасно работают и в Windows, и в Unix без изменений (конечно, в этом помогает и переносимость tkinter). Сценарий PyGadgets даже настраивает `PortableLauncher` для изменения метки в графическом интерфейсе во время запуска.

```
class Launcher(launchmodes.PortableLauncher): # обертка класса запуска
    def announce(self, text):                 # изменяет метку в ГИ
        Info.config(text=text)
```

Мы исследуем эти два и другие клиентские сценарии, такие как PyEdit во второй части книги, после того как приступим к созданию графических интерфейсов в третьей части. Отчасти из-за своей роли в сценарии PyEdit в данном издании книги этот модуль был дополнен функцией, автоматически замещающей символы прямого слеша в пути к файлу символами *обратного слеша*. В PyEdit в некоторых именах файлов используются символы прямого слеша, потому что они допустимы в операциях открытия файлов в системе Windows, но некоторые инструменты запуска программ в Windows требуют использования символов

обратного слеша. В частности, функции `system`, `popen` и `startfile` из модуля `os` требуют использования символов обратного слеша, а функция `spawnv` – нет. Сценарий PyEdit и другие автоматически наследуют исправление путей к файлам в виде функции `fixWindowsPath` за счет импортирования и использования классов из этого модуля. Сценарий PyEdit был изменен так, чтобы устранить влияние этой функции, как неподходящее для данного конкретного случая (смотрите главу 11), но другие клиенты получают это исправление автоматически.

Обратите также внимание, что некоторые из классов в этом примере используют строку пути `sys.executable`, чтобы получить полный путь к выполняемому файлу Python. Отчасти это обусловлено их использованием в сценариях запуска демонстрационных примеров. В предыдущих версиях, до появления атрибута `sys.executable`, эти классы использовали две функции, экспортируемые модулем `Launcher.py`, которые отыскивали выполняемый файл интерпретатора независимо от того, поместил ли пользователь этот путь в переменную окружения `PATH`.

Теперь необходимость в таком поиске отпала. Поскольку я еще буду возвращаться к этому модулю в следующих главах, а также потому, что необходимость такого поиска отпала, – благодаря бесконечному потаканию Python профессиональным желаниям программистов – я отложу бессмысленные педагогические наставления в сторону. (Точка.)

Другие системные инструменты

На этом мы завершаем тур по инструментам системного программирования, имеющимся в языке Python. В этой и в трех предыдущих главах мы познакомились с большинством часто используемых системных инструментов из библиотеки Python. Попутно мы научились использовать их для таких полезных вещей, как запуск программ, обработка каталогов и так далее. Следующая глава завершает исследование этой области представлением примеров использования инструментов, с которыми мы только что познакомились, для реализации сценариев, выполняющих более полезную и практическую работу на системном уровне.

Тем не менее в Python есть и другие системные инструменты, которые появятся в этой книге дальше. Например:

- Сокеты, используемые для обмена данными с другими программами, которые коротко были представлены здесь, встретятся нам снова в главе 10, где будут использоваться в графическом интерфейсе, а полный охват сокетов вы найдете в главе 12.
- Функции выбора, используемые для организации многозадачности, также будут представлены в главе 12, как средство реализации серверов.
- Прием блокировки файлов с помощью функции `os.open`, представленной в главе 4, будет обсуждаться в последующих примерах.

- Регулярные выражения, поиск строк по шаблону, используемый во многих инструментах обработки текста, применяемых в системном администрировании, появятся только в главе 19.

Кроме того, такие приемы, как ветвление и потоки, интенсивно используются в главах, посвященных разработке сценариев для Интернета: смотрите обсуждение многопоточных графических интерфейсов в главах 9 и 10; реализации серверов в главе 12; графические интерфейсы клиентов FTP в главе 13 и пример программы PyMailGUI в главе 14. По пути нам также встретятся высокоуровневые модули Python, такие как `socketserver`, использующие приемы ветвления и потоки выполнения для реализации серверов. Многие инструменты, описанные в этой главе, будут постоянно появляться в дальнейших примерах этой книги – а для чего же еще создаются переносимые библиотеки общего назначения?

Последнее, но не маловажное, что я хотел бы еще раз подчеркнуть: в библиотеке Python есть много других инструментов, которые вообще не фигурируют в данной книге. При наличии сотен модулей в библиотеке и еще большего количества сторонних модулей авторам, пишущим книги по Python, приходится ограничивать себя в отборе тем! Как всегда, напомню о необходимости изучения руководства по этой библиотеке, как в начале, так и на всем протяжении вашей карьеры программиста Python.

6

Законченные системные программы

«Ярость поиска»

Эта глава завершает обзор системных интерфейсов Python и представляет коллекцию более крупных сценариев на языке Python, которые решают практические системные задачи – сравнение и копирование деревьев каталогов, разрезание файлов, поиск файлов и каталогов, тестирование других программ, настройка окружения запускаемых программ и так далее. Примеры в этой главе являются системными утилитами на языке Python, иллюстрирующими типичные решения и приемы программирования, применяемые в этой области, и основное внимание здесь уделяется использованию встроенных инструментов, таких как инструменты обработки файлов и деревьев каталогов.

Главная цель главы состоит в том, чтобы дать вам почувствовать практические сценарии в действии. Размеры этих примеров также дают возможность увидеть действие таких парадигм программирования на языке Python, как объектно-ориентированное программирование (ООП) и повторное использование программного кода. Только в контексте нетривиальных программ, таких как примеры в этой главе, применение подобных инструментов начинает приносить ощутимые плоды. В данной главе вы найдете ответы не только на вопрос «как», но и «почему» – попутно я буду показывать, для удовлетворения каких насущных потребностей создавались сценарии, которые мы будем рассматривать, чтобы помочь вам совместить подробности реализации с контекстом.

Одно предварительное замечание: в этой главе мы будем двигаться вперед очень быстро, а некоторые из представленных здесь примеров предназначены в основном для самостоятельного изучения. Все сценарии хорошо документированы и используют системные инструменты Python, описанные в предыдущих главах, поэтому я не буду подробно описывать весь программный код. Вам следует самостоятельно озна-

комиться с исходными текстами сценариев и поэкспериментировать с ними на своем компьютере, чтобы лучше почувствовать, как комбинировать системные интерфейсы для решения практических задач. Все сценарии включены в состав пакета с примерами для этой книги, и большая их часть работает на всех основных платформах.

Следует также упомянуть, что большую часть этих программ я использую на практике, то есть это не просто примеры, написанные для книги. Они создавались на протяжении нескольких лет и решают самые разнообразные задачи, поэтому их ничто не объединяет, кроме реальности. С другой стороны, эти примеры позволяют показать полезность системных инструментов, продемонстрировать крупные концепции разработки, что невозможно сделать на простых примерах, и наглядно доказать простоту автоматизации системных задач на языке Python переносимым способом. Овладев основами, вы будете сожалеть, что не сделали этого раньше.

Игра: «Найди самый большой файл Python»

Попробуйте быстро ответить на вопрос: «Как называется самый большой файл с программным кодом на языке Python на вашем компьютере?» Этот невинный вопрос был однажды задан мне студентом на курсах, которые я веду. Поскольку я не знал ответ на него, это послужило поводом включить реализацию такого сценария в качестве примера в мой курс обучения, который стал отличной иллюстрацией способов применения системных инструментов Python для решения практических задач. В действительности этот вопрос звучит несколько неопределенно, потому что не совсем понятна область, к которой он относится. Подразумевается ли наибольший файл в каком-то определенном каталоге, в дереве каталогов, в стандартной библиотеке, в пути поиска модулей или вообще на всем жестком диске? Различные области подразумевают различные решения.

Сканирование каталога стандартной библиотеки

Так, в примере 6.1 приводится первое решение, которое отыскивает наибольший файл Python в ограниченной области, – в одном каталоге, но этого вполне достаточно для начала.

Пример 6.1. PP4E\System\Filetools\bigpy-dir.py

```
"""
Отыскивает наибольший файл с исходным программным кодом на языке Python
в единственном каталоге.
Поиск выполняется в каталоге стандартной библиотеки Python для Windows, если
в аргументе командной строки не был указан какой-то другой каталог.
"""
import os, glob, sys
dirname = r'C:\Python31\Lib' if len(sys.argv) == 1 else sys.argv[1]
```

```
allsizes = []
allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])
```

Для обхода файлов в каталоге этот сценарий использует модуль `glob`. Он определяет размеры и имена файлов и сохраняет информацию в списке, который затем сортируется. Поскольку размер является первым элементом кортежей, помещаемых в список, список будет отсортирован по размерам файлов, вследствие чего информация о наибольшем файле окажется в конце списка. Вместо того чтобы сохранять весь список, можно было бы сохранять только текущий наибольший файл, но решение со списком выглядит более гибким. Сценарий сканирует каталог стандартной библиотеки Python в Windows, если в аргументе командной строки ему не был передан другой каталог, и выводит информацию о наибольшем и наименьшем файлах, обнаруженных им:

```
C:\...\PP4E\System\Filetools> bigpy-dir.py
[(0, 'C:\\Python31\\Lib\\build_class.py'), (56, 'C:\\Python31\\Lib\\struct.py')]
[(147086, 'C:\\Python31\\Lib\\turtle.py'), (211238, 'C:\\Python31\\Lib\\decimal.py')]

C:\...\PP4E\System\Filetools> bigpy-dir.py .
[(21, '.\\__init__.py'), (461, '.\\bigpy-dir.py')]
[(1940, '.\\bigext-tree.py'), (2547, '.\\split.py')]

C:\...\PP4E\System\Filetools> bigpy-dir.py ..
[(21, '..\\__init__.py'), (29, '..\\testargv.py')]
[(541, '..\\testargv2.py'), (549, '..\\more.py')]
```

Сканирование дерева каталогов стандартной библиотеки

Решение, предложенное в предыдущем разделе, работает, но совершенно очевидно, представляет частичный ответ на поставленный вопрос – файлы с исходными текстами на языке Python обычно располагаются более чем в одном каталоге. Даже стандартная библиотека содержит множество подкаталогов с модулями, которые могут произвольно вкладываться друг в друга. В действительности нам необходимо реализовать обход всего дерева каталогов. Кроме того, в выводе сценария, приведенном выше, не так-то просто разобраться. Исправить эту проблему нам поможет модуль `pprint` (от «pretty print» – форматированный вывод). Все эти улучшения добавлены в сценарий, представленный в примере 6.2.

Пример 6.2. PP4E\System\Filetools\bigpy-tree.py

```
"""
Отыскивает наибольший файл с исходным программным кодом на языке Python в дереве
каталогов.
Поиск выполняется в каталоге стандартной библиотеки, отображение результатов
выполняется с помощью модуля pprint.
"""
import sys, os, pprint
trace = False
if sys.platform.startswith('win'):
    dirname = r'C:\Python31\Lib'       # Windows
else:
    dirname = '/usr/lib/python'        # Unix, Linux, Cygwin

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    if trace: print(thisDir)
    for filename in filesHere:
        if filename.endswith('.py'):
            if trace: print('...', filename)
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])
```

Для поиска наибольшего файла с программным кодом на языке Python в дереве каталогов эта версия использует `os.walk`. Если вы хотите увидеть, как выполняется обход каталогов, измените значение переменной `trace`. В данной реализации сценарий способен выполнять обход дерева каталогов стандартной библиотеки Python и в Windows, и в Unix-подобных системах:

```
C:\...\PP4E\System\Filetools> bigpy-tree.py
[(0, 'C:\\Python31\\Lib\\build_class.py'),
 (0, 'C:\\Python31\\Lib\\email\\mime\\__init__.py')]
[(211238, 'C:\\Python31\\Lib\\decimal.py'),
 (380582, 'C:\\Python31\\Lib\\pydoc_data\\topics.py')]
```

Сканирование пути поиска модулей

Как и следовало ожидать, сценарий из предыдущего раздела отыскивает наименьший и наибольший файлы в дереве каталогов. Хотя поиск в полном дереве каталогов стандартной библиотеки Python дает более исчерпывающий ответ, тем не менее его никак нельзя признать полным: на компьютере могут находиться дополнительные модули, установленные в другие каталоги, включенные в путь поиска модулей, но

за пределами дерева файлов с исходными текстами на языке Python. Чтобы дать более полный ответ, необходимо выполнить все тот же поиск в дереве каталогов, но при этом следует просмотреть все каталоги, включенные в путь поиска модулей. Это улучшение было добавлено в сценарий, представленный в примере 6.3, – он просматривает все модули Python, доступные для импортирования и располагающиеся непосредственно в пути поиска, а также расположенные во вложенных каталогах пакетов.

Пример 6.3. PP4E\System\Filetools\bigpy-path.py

```
"""
Отыскивает наибольший файл с исходным программным кодом на языке Python,
присутствующий в пути поиска модулей.
Пропускает каталоги, которые уже были просканированы; нормализует пути и регистр
символов, обеспечивая корректность сравнения; включает в выводимые результаты
счетчики строк. Здесь недостаточно использовать os.environ['PYTHONPATH']:
этот список является лишь подмножеством списка sys.path.
"""
import sys, os, pprint
trace = 0                               # 1=каталоги, 2=+файлы

visited = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        if trace > 0: print(thisDir)
        thisDir = os.path.normpath(thisDir)
        fixcase = os.path.normcase(thisDir)
        if fixcase in visited:
            continue
        else:
            visited[fixcase] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                if trace > 1: print('...', filename)
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except os.error:
                    print('skipping', pypath, sys.exc_info()[0])
                else:
                    pylines = len(open(pypath, 'rb').readlines())
                    allsizes.append((pysize, pylines, pypath))

print('By size...')
allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])
```

```
print('By lines...')
allsizes.sort(key=lambda x: x[1])
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])
```

Этот сценарий выполняет обход всех каталогов, включенных в путь поиска, и для каждого из них пытается выполнить поиск в полном дереве подкаталогов. В результате получается тройной вложенный цикл – цикл по элементам пути, цикл по каталогам в дереве очередного элемента и цикл по файлам в каталоге. Так как путь поиска модулей может содержать каталоги, имена которых могут записываться произвольно, кроме всего прочего этот сценарий должен побеспокоиться, чтобы:

- Нормализовать пути к каталогам – исправить символы слеша и точки, чтобы привести имена каталогов к общепринятому виду.
- Нормализовать регистр символов в именах каталогов – привести к нижнему регистру все символы в именах файлов и каталогов в нечувствительном к регистру символов Windows, чтобы свести определение эквивалентности имен каталогов к простой операции сравнения строк, но не изменять регистр символов в Unix, где он имеет значение.
- Выявлять повторы, чтобы избежать повторного сканирования одних и тех же каталогов (один и тот же каталог может оказаться достижимым при сканировании разных элементов, включенных в `sys.path`).
- Пропускать все элементы, похожие на файлы, для которых функция `os.path.getsize` возбуждает исключение (по умолчанию `os.walk` сама молча игнорирует элементы на любом уровне вложенности, которые не являются каталогами).
- Избежать возможных *ошибок декодирования символов Юникода* в содержимом файлов, открывая их для подсчета строк в двоичном режиме. В текстовом режиме выполняется обязательное декодирование содержимого, а некоторые файлы в дереве каталогов библиотеки Python 3.1 не могут быть корректно декодированы в Windows. Перехват ошибок декодирования с помощью инструкции `try` позволил бы предотвратить преждевременное завершение программы, но при этом могли бы быть пропущены потенциальные кандидаты на звание большего или меньшего файла.

В эту версию был добавлен подсчет строк, что может существенно увеличить время работы сценария, но это очень интересный отчетный показатель. Фактически данная версия использует это значение как ключ сортировки, чтобы определить три наибольших и наименьших по количеству строк файла, – эти результаты могут не совпадать с результатами, когда наибольший и наименьший размер определяется по размеру файла в байтах. Ниже приводятся результаты выполнения этого сценария в Python 3.1 на моем компьютере, работающем под управле-

нием Windows 7. Так как результаты зависят от платформы, наличия дополнительных расширений и настроек пути поиска, у вас могут получиться другие наибольший и наименьший файлы в пути sys.path:

```
C:\...\PP4E\System\Filetools> bigpy-path.py
By size...
[(0, 0, 'C:\\Python31\\lib\\build_class.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\mime\\__init__.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\test\\__init__.py')]
[(161613, 3754, 'C:\\Python31\\lib\\tkinter\\__init__.py'),
 (211238, 5768, 'C:\\Python31\\lib\\decimal.py'),
 (380582, 78, 'C:\\Python31\\lib\\pydoc_data\\topics.py')]
By lines...
[(0, 0, 'C:\\Python31\\lib\\build_class.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\mime\\__init__.py'),
 (0, 0, 'C:\\Python31\\lib\\email\\test\\__init__.py')]
[(147086, 4132, 'C:\\Python31\\lib\\turtle.py'),
 (150069, 4268, 'C:\\Python31\\lib\\test\\test_descr.py'),
 (211238, 5768, 'C:\\Python31\\lib\\decimal.py')]
```

И снова, если вам интересно увидеть, как выполняется обход каталогов, измените значение переменной trace. Как видите, результаты поиска наибольшего файла по размеру в байтах и по количеству строк отличаются. Это несоответствие, вероятно, мы должны тщательно обсудить на нашей следующей встрече.

Сканирование всего компьютера

Наконец, несмотря на то, что путь поиска модулей обычно включает все исходные файлы Python, доступные для импортирования на вашем компьютере, тем не менее и этот ответ может оказаться неполным. С технической точки зрения, эта версия проверяет только модули – файлы с исходным программным кодом на языке Python, которые запускаются как самостоятельные сценарии, могут не включаться в путь поиска модулей. Кроме того, путь поиска модулей в некоторых сценариях может изменяться вручную прямо во время выполнения (например, прямым изменением списка sys.path в сценариях, выполняющихся на веб-сервере), чтобы включить в него дополнительные каталоги, которые недоступны для сценария в примере 6.3.

В конечном счете, чтобы отыскать наибольший исходный файл на компьютере, необходимо просканировать весь жесткий диск; эта возможность *почти* полностью поддерживается сценарием в примере 6.2. Нам нужно лишь передать ему в аргументе имя корневого каталога и добавить в него некоторые улучшения из версии, сканирующей путь поиска модулей (позволяющие избежать повторного сканирования одних и тех же каталогов в случае поиска по всему компьютеру, пропускать ошибки и подсчитывать строки, если нам не жалко на это времени). В примере 6.4 приводится реализация такого универсального сценария

сканирования дерева каталогов, снабженного усовершенствованиями, необходимыми для сканирования всего диска.

Пример 6.4. PP4E\System\Filetools\bigext-tree.py

```
"""
Отыскивает наибольший файл заданного типа в произвольном дереве каталогов.
Пропускает каталоги, которые уже были просканированы; перехватывает ошибки;
добавляет возможность вывода трассировки поиска и подсчета строк.
Кроме того, использует множества, итераторы файлов и генераторы, чтобы избежать
загрузки содержимого файлов целиком, и пытается обойти проблемы, возникающие при
выводе недекодируемых имен файлов/каталогов.
"""
import os, pprint
from sys import argv, exc_info

trace = 1                                   # 0=выкл., 1=каталоги, 2=+файлы
dirname, extname = os.curdir, '.py'         # по умолчанию файлы .py в cwd
if len(argv) > 1: dirname = argv[1]         # например: C:\, C:\Python31\Lib
if len(argv) > 2: extname = argv[2]         # например: .pyw, .txt
if len(argv) > 3: trace   = int(argv[3])    # например: ". .py 2"

def tryprint(arg):
    try:
        print(arg)                          # непечатаемое имя файла?
    except UnicodeEncodeError:
        print(arg.encode())                 # вывести как строку байтов

visited  = set()
allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    if trace: tryprint(thisDir)
    thisDir = os.path.normpath(thisDir)
    fixname = os.path.normcase(thisDir)
    if fixname in visited:
        if trace: tryprint('skipping ' + thisDir)
    else:
        visited.add(fixname)
        for filename in filesHere:
            if filename.endswith(extname):
                if trace > 1: tryprint('+++' + filename)
                fullname = os.path.join(thisDir, filename)
                try:
                    bytesize = os.path.getsize(fullname)
                    linesize = sum(+1 for line in open(fullname, 'rb'))
                except Exception:
                    print('error', exc_info()[0])
                else:
                    allsizes.append((bytesize, linesize, fullname))

for (title, key) in [('bytes', 0), ('lines', 1)]:
```

```
    print('\nBy %s...' % title)
    allsizes.sort(key=lambda x: x[key])
    pprint.pprint(allsizes[:3])
    pprint.pprint(allsizes[-3:])
```

В отличие от предыдущей версии, эта позволяет выполнять поиск файлов с определенными расширениями и в определенных каталогах. По умолчанию производится поиск файлов Python в текущем рабочем каталоге:

```
C:\...\PP4E\System\Filetools> bigext-tree.py
.

By bytes...
[(21, 1, '.\\__init__.py'),
 (461, 17, '.\\bigpy-dir.py'),
 (818, 25, '.\\bigpy-tree.py')]
[(1696, 48, '.\\join.py'),
 (1940, 49, '.\\bigext-tree.py'),
 (2547, 57, '.\\split.py')]

By lines...
[(21, 1, '.\\__init__.py'),
 (461, 17, '.\\bigpy-dir.py'),
 (818, 25, '.\\bigpy-tree.py')]
[(1696, 48, '.\\join.py'),
 (1940, 49, '.\\bigext-tree.py'),
 (2547, 57, '.\\split.py')]
```

Настраивая работу сценария, можно задать имя каталога, расширение искомых файлов и уровень подробности вывода трассировочной информации (уровень 0 запрещает трассировку, при уровне 1 (по умолчанию) выводятся имена сканируемых каталогов):

```
C:\...\PP4E\System\Filetools> bigext-tree.py .. .py 0

By bytes...
[(21, 1, '..\\__init__.py'),
 (21, 1, '..\\Filetools\\__init__.py'),
 (28, 1, '..\\Streams\\hello-out.py')]
[(2278, 67, '..\\Processes\\multi2.py'),
 (2547, 57, '..\\Filetools\\split.py'),
 (4361, 105, '..\\Tester\\tester.py')]

By lines...
[(21, 1, '..\\__init__.py'),
 (21, 1, '..\\Filetools\\__init__.py'),
 (28, 1, '..\\Streams\\hello-out.py')]
[(2547, 57, '..\\Filetools\\split.py'),
 (2278, 67, '..\\Processes\\multi2.py'),
 (4361, 105, '..\\Tester\\tester.py')]
```

Кроме того, этот сценарий поволяет выполнять поиск файлов разных типов. Ниже приводятся результаты поиска наибольшего и наименьшего текстового файла, начиная с каталога уровнем выше текущего (эти результаты имели место на тот момент времени, когда я запускал сценарий):

```
C:\...\PP4E\System\Filetools> bigext-tree.py .. .txt 1
..
..\Environment
..\Filetools
..\Processes
..\Streams
..\Tester
..\Tester\Args
..\Tester\Errors
..\Tester\Inputs
..\Tester\Outputs
..\Tester\Scripts
..\Tester\xxold
..\Threads

By bytes...
[(4, 2, '..\\Streams\\input.txt'),
 (13, 1, '..\\Streams\\hello-in.txt'),
 (20, 4, '..\\Streams\\data.txt')]
[(104, 4, '..\\Streams\\output.txt'),
 (172, 3, '..\\Tester\\xxold\\README.txt.txt'),
 (435, 4, '..\\Filetools\\temp.txt')]

By lines...
[(13, 1, '..\\Streams\\hello-in.txt'),
 (22, 1, '..\\spam.txt'),
 (4, 2, '..\\Streams\\input.txt')]
[(20, 4, '..\\Streams\\data.txt'),
 (104, 4, '..\\Streams\\output.txt'),
 (435, 4, '..\\Filetools\\temp.txt')]
```

А чтобы выполнить поиск по всей системе, достаточно просто передать сценарию имя корневого каталога (в Unix-подобных системах вместо `C:\` используйте `/`) и расширение файлов (по умолчанию используется расширение *.py*). Итак, победитель... (только не заключайте никакие пари):

```
C:\...\PP4E\dev\Examples\PP4E\System\Filetools> bigext-tree.py C:\
C:\
C:\$Recycle.Bin
C:\$Recycle.Bin\S-1-5-21-3951091421-2436271001-910485044-1004
C:\cygwin
C:\cygwin\bin
C:\cygwin\cygdrive
C:\cygwin\dev
C:\cygwin\dev\mqueue
```

```
C:\cygwin\dev\shm
C:\cygwin\etc
```

...МНОГО строк опущено...

```
By bytes...
[(0, 0, 'C:\\cygwin\\...\\python31\\Python-3.1.1\\Lib\\build_class.py'),
 (0, 0, 'C:\\cygwin\\...\\python31\\Python-3.1.1\\Lib\\email\\mime\\__init__.
py'),
 (0, 0, 'C:\\cygwin\\...\\python31\\Python-3.1.1\\Lib\\email\\test\\__init__.
py')]
[(380582, 78, 'C:\\Python31\\Lib\\pydoc_data\\topics.py'),
 (398157, 83, 'C:\\...\\Install\\Source\\Python-2.6\\Lib\\pydoc_topics.py'),
 (412434, 83, 'C:\\Python26\\Lib\\pydoc_topics.py')]

By lines...
[(0, 0, 'C:\\cygwin\\...\\python31\\Python-3.1.1\\Lib\\build_class.py'),
 (0, 0, 'C:\\cygwin\\...\\python31\\Python-3.1.1\\Lib\\email\\mime\\__init__.
py'),
 (0, 0, 'C:\\cygwin\\...\\python31\\Python-3.1.1\\Lib\\email\\test\\__init__.
py')]
[(204107, 5589, 'C:\\...\Install\\Source\\Python-3.0\\Lib\\decimal.py'),
 (205470, 5768, 'C:\\cygwin\\...\\python31\\Python-3.1.1\\Lib\\decimal.py'),
 (211238, 5768, 'C:\\Python31\\Lib\\decimal.py')]
```

Логика трассировки построена так, что позволяет следить за тем, как выполняется обход каталогов. Я сократил список каталогов в этом примере, чтобы не перегружать его лишней информацией (и уместить на страницу). Для выполнения этой команды может потребоваться достаточно продолжительное время. На моем нетбуке, работающем, как это ни печально, под управлением Windows 7, потребовалось 11 минут, чтобы просканировать жесткий диск, содержащий примерно 59 Гбайт данных, 200K файлов и 25K каталогов, при невысокой нагрузке на систему (8 минут – при отключенной трассировке, и полчаса, когда было запущено множество других приложений). Однако этот сценарий дает самый исчерпывающий ответ на поставленный вопрос.

Это решение настолько полное, насколько позволяет пространство в книге. Ради интереса подумайте о возможности сканирования нескольких дисков; о том, что исходные файлы Python могут находиться в zip-архивах, как в пути поиска модулей, так и за его пределами (`os.walk` просто игнорирует zip-файлы в примере 6.3). Кроме того, файлы с исходными текстами могут иметь разные расширения – файлы с расширением *.pyw* подавляют вывод окна консоли в Windows, а файлы сценариев верхнего уровня могут иметь произвольные расширения. Фактически имена файлов сценариев верхнего уровня вообще могут не иметь расширения и при этом оставаться файлами с исходными текстами на языке Python. Кроме того, некоторые модули, доступные для импортирования, не являясь файлами с исходными текстами на языке Python, могут присутствовать в формате скомпилированных двоичных файлов или быть статически связаны с выполняемым файлом интер-

претатора Python. В интересах экономии места мы оставим эти варианты (довольно сложные в реализации!) расширения процедуры поиска, как упражнение для самостоятельного решения.

Вывод имен файлов с символами Юникода

Одна тонкость, прежде чем двинуться дальше: обратите внимание на, казалось бы, излишнюю конструкцию обработки исключений в функции `tryprint` из примера 6.4. Когда я в первый раз попытался просканировать весь диск с помощью сценария, приведенного в предыдущем разделе, этот сценарий завершился с ошибкой декодирования символов Юникода при попытке вывести имя каталога сохраненной веб-страницы. Добавление обработчика исключения позволило просто пропустить этот каталог.

Этот случай наглядно демонстрирует тонкую, но имеющую большое практическое значение проблему: ориентированность Python 3.X на Юникод распространяется и на имена файлов, даже в случае простого их вывода. Как мы узнали в главе 4, имена файлов могут содержать произвольный текст, поэтому функция `os.listdir` способна возвращать имена файлов в двух различных представлениях – она возвращает декодированные строки Юникода, если получает аргумент типа `str`, и кодированную строку байтов, если получает аргумент типа `bytes`:

```
>>> import os
>>> os.listdir('.')[:4]
['bigext-tree.py', 'bigpy-dir.py', 'bigpy-path.py', 'bigpy-tree.py']

>>> os.listdir(b'.')[:4]
[b'bigext-tree.py', b'bigpy-dir.py', b'bigpy-path.py', b'bigpy-tree.py']
```

Обе функции, `os.walk` (используется в примере 6.4) и `glob.glob`, наследуют это поведение при возвращении имен файлов и каталогов, потому что внутри они используют функцию `os.listdir`. Во всех этих функциях передача строки байтов в аргументе подавляет декодирование символов Юникода в именах файлов и каталогов. Передача обычной строки предполагает, что имена файлов могут быть декодированы при применении кодировки, используемой файловой системой.

Причина, по которой данная особенность может иметь важное значение для сценария из этого раздела, состоит в том, что версия, выполняющая поиск по всему жесткому диску, в конце концов может столкнуться с недекодируемыми именами файлов (например, ранее сохраненная веб-страница с необычным именем), что приводит к возбуждению исключения при попытке вывести их с помощью функции `print`. Ниже приводится упрощенный пример, выполняемый в окне консоли Windows, который воспроизводит ошибку:

```
>>> root = r'C:\py3000'
>>> for (dir, subs, files) in os.walk(root): print(dir)
...
```

```
C:\py3000
C:\py3000\FutureProofPython - PythonInfo Wiki_files
C:\py3000\Oakwinter_com Code » Porting setuptools to py3k_files
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python31\lib\encodings\cp437.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_map)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u2019' in position
45: character maps to <undefined>
```

(UnicodeDecodeError: кодек 'charmap' не может преобразовать символ '\u2019' в позиции 45: отображается в символ <undefined>)

Один из способов выхода из этого затруднительного положения состоит в том, чтобы передавать имя корневого каталога в виде строки типа bytes, – это подавляет выполнение операции декодирования имен файлов в функции os.listdir, вызываемой функцией os.walk, и эффективно ограничивает область действия последующих операций вывода простыми байтами. Так как в этом случае операциям вывода не приходится иметь дело с кодировками, они выполняются без ошибок. Декодирование строк в байты вручную перед выводом тоже может помочь, но результаты получаются немного другими:

```
>>> root.encode()
b'C:\\py3000'

>>> for (dir, subs, files) in os.walk(root.encode()): print(dir)
...
b'C:\\py3000'
b'C:\\py3000\\FutureProofPython - PythonInfo Wiki_files'
b'C:\\py3000\\Oakwinter_com Code \xbb Porting setuptools to py3k_files'
b'C:\\py3000\\What\x92s New in Python 3_0 \x97 Python Documentation'

>>> for (dir, subs, files) in os.walk(root): print(dir.encode())
...
b'C:\\py3000'
b'C:\\py3000\\FutureProofPython - PythonInfo Wiki_files'
b'C:\\py3000\\Oakwinter_com Code \xc2\xbb Porting setuptools to py3k_files'
b'C:\\py3000\\What\xe2\x80\x99s New in Python 3_0 \xe2\x80\x94 Python
Documentation'
```

К сожалению, при любом подходе все имена каталогов, которые выводятся в процессе обхода, отображаются как непонятные строки байтов. Чтобы не отказываться от обычных строк, обеспечивающих более высокую удобочитаемость, я выбрал вариант реализации с обработчиком исключений. Это позволяет полностью исключить проблемы:

```
>>> for (dir, subs, files) in os.walk(root):
...     try:
...         print(dir)
...     except UnicodeEncodeError:
```

```
...         print(dir.encode()) # или просто пропустить, если encode
...                             # может потерпеть неудачу
C:\py3000
C:\py3000\FutureProofPython - PythonInfo Wiki_files
C:\py3000\Oakwinter_com Code » Porting setuptools to py3k_files
b'C:\\py3000\\What\xe2\x80\x99s New in Python 3_0 \xe2\x80\x94 Python
Documentation'
```

Странно, но похоже, что ошибка связана скорее с выводом, чем с кодированием символов Юникода в именах файлов, – поскольку при работе с именами файлов никаких ошибок не возникает, пока не предпринимаются попытки вывести их, следовательно, они были успешно декодированы в момент первоначального преобразования их в строки. Именно поэтому достаточно обернуть в инструкции `try` вызовы функции `print`, в противном случае ошибка возникала бы раньше.

Более того, эта ошибка не возникает, если стандартный поток вывода сценария перенаправить в файл, на уровне командной оболочки (`bigexttree.py c:\ > out`) или в вызове самой функции `print` (`print(dir, file=F)`). В последнем случае чтение выходного файла должно выполняться в двоичном режиме, так как попытка вывести в окно консоли содержимое файла, открытого в текстовом режиме, приведет к той же ошибке (и снова, ошибка не возникает, пока не будет предпринята попытка вывода). Фактически программный код, который терпит неудачу при запуске в окне программы Командная строка (Command Prompt) в Windows, работает без ошибок в графическом интерфейсе IDLE, на той же самой платформе, – графический интерфейс IDLE, реализованный на основе библиотеки tkinter, выполняет обработку отображаемых символов, что не делается, когда символы выводятся в поток стандартного вывода, подключенный к окну терминала:

```
>>> import os # запускайте в IDLE (в графическом интерфейсе на базе tkinter),
             # а не в системной командной оболочке
>>> root = r'C:\py3000'
>>> for (dir, subs, files) in os.walk(root): print(dir)

C:\py3000
C:\py3000\FutureProofPython - PythonInfo Wiki_files
C:\py3000\Oakwinter_com Code » Porting setuptools to py3k_files
C:\py3000\What's New in Python 3_0 – Python Documentation_files
```

Другими словами, исключение возникает только при выводе в окно командной оболочки, уже после того, как будет создана строка с именем файла. Это свидетельствует о выполнении дополнительных преобразований в подсистеме вывода Python, что никак не связано с наличием символов Юникода в именах файлов. Однако, за неимением места для дальнейших рассуждений, нам придется удовлетвориться констатацией факта, что наш обработчик исключений позволяет обойти проблему вывода. Однако вам следует помнить о проблемах декодирования сим-

волов Юникода в именах файлов – на некоторых платформах может потребоваться в этом сценарии передавать функции `os.walk` строки байтов, чтобы предотвратить ошибки декодирования имен файлов.[1]

Механизм поддержки Юникода в версии 3.1 все еще остается относительно новым, поэтому обязательно проверяйте наличие подобных ошибок в вашей системе и в вашей версии Python. Кроме того, дополнительную информацию по работе с именами файлов, содержащими символы Юникода, можно найти в руководствах по языку Python, а о поддержке Юникода вообще – в книге «Изучаем Python». Как отмечалось выше, сценарии должны открывать текстовые файлы в двоичном режиме, если они могут содержать недекодируемое *содержимое*. Может показаться странным, что проблемы, связанные с поддержкой Юникода, могут так отражаться на простейших операциях вывода, как в данном примере, но такова жизнь в прекрасном мире Юникода. При этом никаких проблем, связанных с Юникодом, не возникает в большой доле реальных сценариев, включая те, что мы будем рассматривать в следующем разделе.

Разрезание и объединение файлов

Мои дети, как и многие другие, проводили массу времени в Интернете, пока не повзрослели. Насколько мне известно, в наши дни это считалось стоящим занятием. Среди представителей этого нового поколения компьютерные фанаты и гуру пользуются, вероятно, таким же уважением, как когда-то рок-звезды для моего поколения. Когда дети скрываются в своих комнатах, они, скорее всего, возятся с компьютерами, а не осваивают гитарные аккорды (по крайней мере, некоторые из них). Пожалуй, это более здоровое занятие, чем некоторые утехи моей собственной растраченной юности, но это уже тема для книги иного сорта.

Несмотря на риторику ученых мужей, вещающих о доступных новому поколению потенциальных возможностях Интернета, которые трудно было даже представить их родителям, мои дети больше времени все-таки потратили на компьютерные игры. Чтобы получить новые игры,

[1] Сходная проблема, связанная с функцией `print`, описывается в главе 14, где предлагается обходное решение, позволяющее предотвратить аварийное завершение программы в случае вывода трассировочной информации в поток стандартного вывода порожденными программами. В отличие от проблемы, описываемой здесь, та проблема, похоже, не связана с выводом символов Юникода, которые могут оказаться непечатаемыми в окне командной оболочки, а отражает еще один недостаток механизма вывода в стандартный поток вывода в версии Python 3.1, который может быть исправлен к моменту, когда вы будете читать эти строки. Смотрите также описание переменной окружения `PYTHONIOENCODING`, с помощью которой можно переопределить кодировку, используемую по умолчанию для стандартного потока вывода.

им в то время приходилось загружать их на общий компьютер, подключенный к Интернету, и затем переносить на свои компьютеры для установки у себя. (Их компьютеры были подключены к Интернету позднее, по причинам, список которых у каждых родителей может быть свой.)

Проблема в том, что файлы с играми далеко не маленькие. Обычно они не умещались ни на гибких дисках, ни на флешках того времени, а запись на CD или DVD отнимала драгоценное время, которое можно было бы потратить на игру. Если бы все компьютеры у меня дома работали под управлением Linux, это не было бы проблемой. Для Unix существуют программы командной строки, позволяющие разрезать файлы на части, достаточно маленькие, чтобы уместиться на переносном устройстве (команда `split`), и программы для обратного объединения фрагментов (команда `cat`). Однако поскольку у нас дома были самые разные компьютеры, нам необходимо было более переносимое решение.[1]

Разрезание файлов переносимым способом

Так как на всех компьютерах в моем доме стоит Python, на помощь приходит простой переносимый сценарий на этом языке. В примере 6.5 приводится программа на языке Python, которая распределяет содержимое одного файла по группе файлов меньшего размера, которые сохраняются в каталоге.

Пример 6.5. PP4E\System\Filetools\split.py

```
#!/usr/bin/python
"""
###############################################################################
разрезает файл на несколько частей; сценарий join.py объединяет эти части в один
файл; данный сценарий является настраиваемой версией стандартной команды split
в Unix; поскольку сценарий написан на языке Python, он с тем же успехом может
использоваться в Windows и легко может быть модифицирован; благодаря тому, что
он экспортирует функцию, его логику можно импортировать и повторно использовать
в других приложениях;
###############################################################################
"""

import sys, os
kilobytes = 1024
```

[1] Необходимо отметить, что эту историю я рассказывал еще во втором издании книги, которое вышло в 2000 году. За последние десять лет гибкие диски практически закончили путь, который уже прошли параллельные порты и динозавры. Кроме того, запись CD или DVD выполняется не так долго, как прежде; в настоящее время появились новые возможности, такие как флешки огромного объема, беспроводные сети и даже простая электронная почта; и, естественно, у меня дома теперь стоят более совершенные компьютеры. А что касается моих детей – некоторые из них уже выросли (хотя и сохранили некоторую обратную совместимость с ними прежними).

```
megabytes = kilobytes * 1000
chunksize = int(1.4 * megabytes)      # по умолчанию: примерный размер дискеты

def split(fromfile, todir, chunksize=chunksize):
    if not os.path.exists(todir):       # ошибки обрабатывает вызвавший
        os.mkdir(todir)                 # создать каталог для фрагментов
    else:
        for fname in os.listdir(todir): # удалить все существующие файлы
            os.remove(os.path.join(todir, fname))
    partnum = 0
    input = open(fromfile, 'rb')        # двоичный режим: без декодирования и
                                        # без преобразования символов конца
                                        # строки
    while True:                         # eof = прочтена пустая строка
        chunk = input.read(chunksize)   # прочитать кусок <= chunksize
        if not chunk: break
        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 в join невозможна,
    return partnum                      # если будет 5 цифр

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 = True
            fromfile = input('File to be split? ') # ввод данных, если
                                                   # запущен щелчком мыши
            todir = input('Directory to store part files? ')
        else:
            interactive = False
            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_info()[0], sys.exc_info()[1])
        else:
            print('Split finished:', parts, 'parts are in', absto)
        if interactive: input('Press Enter key') # пауза, если сценарий
                                                 # запущен щелчком мыши
```

По умолчанию этот сценарий разрезает исходный файл на фрагменты, примерно равные размеру дискеты, – идеальные для перемещения больших файлов между не связанными между собой компьютерами. Самое важное здесь – это полностью переносимый программный код; данный сценарий будет работать на любом компьютере, где нет своей встроенной программы для разрезания файлов. Главное, чтобы на компьютере был установлен интерпретатор Python. Ниже приводится пример использования этого сценария для разрезания самоустанавливающегося выполняемого файла Python 3.1 в Windows в текущем каталоге (для экономии места я опустил некоторые строки, которые выводит команда `dir`; в Unix воспользуйтесь командой `ls -l`):

```
C:\temp> cd C:\temp
C:\temp> dir python-3.1.msi
...часть строк опущена...
06/27/2009  04:53 PM        13,814,272 python-3.1.msi
               1 File(s)     13,814,272 bytes
               0 Dir(s)  188,826,189,824 bytes free

C:\temp> python C:\...\PP4E\System\Filetools\split.py -help
Use: split.py [file-to-split target-dir [chunksize]]

C:\temp> python C:\...\P4E\System\Filetools\split.py python-3.1.msi pysplit
Splitting C:\temp\python-3.1.msi to C:\temp\pysplit by 1433600
Split finished: 10 parts are in C:\temp\pysplit

C:\temp> dir pysplit
...часть строк опущена...
02/21/2010  11:13 AM    <DIR>          .
02/21/2010  11:13 AM    <DIR>          ..
02/21/2010  11:13 AM         1,433,600 part0001
02/21/2010  11:13 AM         1,433,600 part0002
02/21/2010  11:13 AM         1,433,600 part0003
02/21/2010  11:13 AM         1,433,600 part0004
02/21/2010  11:13 AM         1,433,600 part0005
02/21/2010  11:13 AM         1,433,600 part0006
02/21/2010  11:13 AM         1,433,600 part0007
02/21/2010  11:13 AM         1,433,600 part0008
02/21/2010  11:13 AM         1,433,600 part0009
02/21/2010  11:13 AM           911,872 part0010
              10 File(s)     13,814,272 bytes
               2 Dir(s)  188,812,328,960 bytes free
```

Каждый из созданных здесь файлов фрагментов представляет один двоичный кусок файла *python-3.1.msi*, достаточно маленький, чтобы поместиться на одной дискете. Действительно, если сложить вместе размеры созданных фрагментов, показанные командой `dir`, то получится точно то же число байтов, что и в оригинальном файле. Прежде чем пытаться снова сложить вместе эти файлы, рассмотрим несколько примечаний, касающихся сценария.

Режимы работы

Данный сценарий может получать свои параметры в *интерактивном* режиме или в режиме *командной строки* – он проверяет количество аргументов командной строки, чтобы определить, в каком режиме используется. В режиме командной строки сценарию передаются имя файла, который нужно разрезать, и каталог для сохранения фрагментов. При этом с помощью необязательного третьего аргумента можно переопределить установленный по умолчанию размер одного фрагмента.

В интерактивном режиме сценарий запрашивает имя файла и каталог для сохранения фрагментов в окне консоли с помощью функции `input` и перед завершением делает остановку, ожидая нажатия клавиши. Этот режим используется, когда программа запускается щелчком на значке файла, – в Windows параметры вводятся во всплывающем окне DOS, которое в этом случае не исчезает автоматически. Сценарий также показывает абсолютные пути для своих параметров (пропуская их через `os.path.abspath`), потому что в интерактивном режиме они могут быть не очевидны.

Двоичный режим доступа к файлам

Этот сценарий открывает входные и выходные файлы в двоичном режиме (`rb`, `wb`), потому что такие файлы, как выполняемые или аудиофайлы, должны обрабатываться переносимым способом, не как текст. В главе 4 мы узнали, что в Windows при работе с текстовыми файлами символы конца строки `\r\n` автоматически отображаются в `\n` при вводе, и `\n` отображается в `\r\n` при выводе. При работе с двоичными данными было бы нежелательно, чтобы `\r` исчезали при чтении и ненужные символы `\r` попадали бы при записи в выходной файл. Для файлов, открываемых в двоичном режиме в Windows, такая трансформация символа `\r` не производится, и искажения данных не происходит.

Кроме того, в Python 3.X при работе с файлами в двоичном режиме данные в сценарии представляются в виде объектов `bytes`, а не в виде кодированного текста `str`. При этом нам даже не приходится предусматривать выполнение каких-то особых действий – операции по обработке файлов, реализованные в сценарии, действуют одинаково и в Python 3.X, и в Python 2.X. Фактически двоичный режим является обязательным для версии 3.X, потому что данные, записываемые в выходной файл, вообще могут не быть текстом – текстовый режим необходим в версии 3.X, только когда приходится выполнять декодирование содержимого файла, которое может приводить к ошибкам при работе с настоящими двоичными файлами и с текстовыми файлами, полученными из других систем. Операция записи в двоичном режиме принимает объект типа `bytes` и подавляет кодирование символов Юникода, а также преобразование символов конца строки.

Закрытие файлов вручную

Этот сценарий также заботится о том, чтобы вручную закрыть используемые им файлы. Как мы видели в главе 4, часто это можно сделать одной строкой: `open(partname, 'wb').write(chunk)`. Эта более короткая форма использует тот факт, что в текущей реализации Python файлы автоматически закрываются при уничтожении объектов файлов (то есть при утилизации их на этапе сборки 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 какой-либо долгоживущей переменной. Аналогично при выходе из функции `split` уничтожается объект файла `input`.

Однако есть вероятность, что такой режим автоматического закрытия в будущем может измениться. Более того, в Jython – реализации Python на языке Java – объекты, на которые нет ссылок, не уничтожаются с такой же поспешностью, как в стандартной реализации Python. Если сейчас или в будущем вам может потребоваться перейти на Java, а ваш сценарий в состоянии создать много файлов за короткий промежуток времени и, возможно, будет выполняться в системе, которая ограничивает количество открытых файлов в каждой программе, закрывайте файлы вручную. Поскольку функция `split` в этом модуле задумана как универсальный инструмент, в ней учтены варианты такого наихудшего развития событий. В главе 4 также упоминался менеджер контекста файла и инструкция `with` – они обеспечивают альтернативный способ гарантированного закрытия файлов.

Соединение файлов переносимым образом

Вернемся к перемещению больших файлов по дому: после загрузки больших файлов игровых программ вы можете воспользоваться предыдущим сценарием *split.py*, щелкнув на нем в окне Проводника Windows (Windows Explorer) и введя имена файлов. После разрезания просто скопируйте каждый файл фрагмента на отдельную дискету (или на более современный носитель), перейдите с дискетами к требуемому компьютеру и скопируйте файлы фрагментов с дискет. Затем щелкните на файле сценария из примера 6.6 (или запустите его другим способом), чтобы вновь объединить фрагменты.

Пример 6.6. PP4E\System\Filetools\join.py

```
#!/usr/bin/python
"""
###############################################################################
объединяет все файлы фрагментов, имеющиеся в каталоге и созданные с помощью
сценария split.py, воссоздавая первоначальный файл.
По своему действию этот сценарий напоминает команду 'cat fromdir/* > tofile'
в Unix, но данная реализация более переносимая и настраиваемая; сценарий
```

```
экспортирует операцию объединения в виде функции, доступной для многократного
использования. Зависит от порядка сортировки имен файлов, поэтому все они
должны быть одинаковой длины. Сценарии разрезания/объединения можно дополнить
возможностью вывода диалога с графическим интерфейсом tkinter, позволяющего
выбирать файлы.
###############################################################################
“””

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’)
        while True:
            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 = True
            fromdir = input(‘Directory containing part files? ‘)
            tofile = input(‘Name of file to be recreated? ‘)
        else:
            interactive = False
            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_info()[0], sys.exc_info()[1])
        else:
            print(‘Join complete: see’, absto)
        if interactive: input(‘Press Enter key’) # пауза, если сценарий
                                                 # запущен щелчком мыши
```

Ниже приводится пример объединения файлов фрагментов в Windows, созданных нами минуту назад. После выполнения сценария `join` вам

может потребоваться воспользоваться какими-нибудь другими утилитами, такими как zip, gzip или tar, чтобы распаковать архивный файл, если он поставлялся не исполняемым, но в любом случае загруженный файл будет готов к дальнейшему использованию:[1]

```
C:\temp> python C:\...\PP4E\System\Filetools\join.py -help
Use: join.py [from-dir-name to-file-name]

C:\temp> python C:\...\PP4E\System\Filetools\join.py pysplit mypy31.msi
Joining C:\temp\pysplit to make C:\temp\mypy31.msi
Join complete: see C:\temp\mypy31.msi

C:\temp> dir *.msi
...часть строк опущена...
02/21/2010  11:21 AM        13,814,272 mypy31.msi
06/27/2009  04:53 PM        13,814,272 python-3.1.msi
               2 File(s)     27,628,544 bytes
               0 Dir(s)  188,798,611,456 bytes free

C:\temp> fc /b mypy31.msi python-3.1.msi
Comparing files mypy31.msi and PYTHON-3.1.MSI
FC: no differences encountered
```

Чтобы выбрать все части файла, присутствующие в каталоге, сценарий join использует функцию os.listdir и сортирует список имен файлов, чтобы расположить части в правильном порядке. Получается точная байт-в-байт копия исходного файла (что проверяется выше командой DOS fc; в Unix используйте команду cmp).

Конечно, часть этого процесса выполняется вручную (я еще не придумал, как запрограммировать этап «перехода с дискетами к другому компьютеру»), но с помощью сценариев split и join перемещение больших файлов становится быстрым и простым. Так как этот сценарий является также переносимым программным кодом Python, он выполняется на любой платформе, на которую может понадобиться перенести разрезанные файлы. Например, у меня дома есть компьютеры, работающие под управлением не только Windows, но и Linux; а так как сценарий выполняется на любой из платформ, у моих игроков не возникает

[1] Как оказывается, команды zip, gzip и tar также можно заменить программным кодом на языке Python. В стандартной библиотеке имеется модуль gzip, который предоставляет инструменты чтения и записи файлов gzip, имена которых обычно имеют расширение *.gz*. 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 последних версиях Python присутствуют также модуль zipfile для обработки архивов в формате ZIP (zip – это формат архивных файлов и сжатия, а gzip – это алгоритм сжатия) и модуль tarfile, позволяющий сценариям работать с tar-архивами. Подробности ищите в руководстве по библиотеке Python.

проблем. Однако, прежде чем двинуться дальше, рассмотрим несколько особенностей реализации сценария `join`:

Чтение файлов блоками целиком

Прежде всего, обратите внимание, что сценарий работает с файлами в двоичном режиме, а также читает файлы фрагментов блоками размером в 1 Кбайт. Значение `readsize` (размер блоков, читаемых из входного файла) не имеет никакого отношения к `chunksize` в сценарии *split.py* (общий размер каждого выходного файла). Как было показано в главе 4, каждый файл фрагмента можно было бы прочесть сразу целиком: `output.write(open(filepath, 'rb').read())`. Недостаток такого приема в том, что при этом весь файл целиком загружается в оперативную память. Например, при чтении файла фрагмента размером 1.4 Мбайт в память целиком в ней будет создана строка размером 1.4 Мбайт, содержащая все байты файла. Поскольку сценарий `split` разрешает пользователям указывать еще более крупные размеры фрагментов, сценарий `join` ожидает худшего и читает содержимое файлов блоками ограниченного размера. Полная надежность была бы обеспечена, если бы сценарий `split` также читал исходный файл меньшими порциями, но на практике в этом нет необходимости (напомню, что в процессе выполнения программы интерпретатор автоматически утилизирует строки, на которые нет ни одной ссылки, поэтому данная реализация не так расточительна, как могло бы показаться).

Сортировка имен файлов

Если внимательно изучить реализацию сценария, можно заметить, что порядок объединения полностью зависит от порядка сортировки имен файлов в каталоге с файлами фрагментов. Так как сценарий объединения просто вызывает метод `sort` списка имен файлов, возвращаемого функцией `os.listdir`, он подразумевает, что при разрезании создаются файлы с именами одинаковой длины и имеющими один и тот же формат. Чтобы удовлетворить это требование, сценарий `split` использует выражение форматирования (`'part%04d'`), которое добавляет незначащие нули и тем самым обеспечивает присутствие одинакового количества цифр (четырех) в именах файлов. Наличие ведущих нулей в маленьких числах гарантирует, что имена файлов фрагментов будут отсортированы правильно.

При желании можно было бы извлекать цифры из имен файлов, преобразовывать их в значения `int` и выполнять числовую сортровку, воспользовавшись аргументом `keys` метода `sort` списков, но в этом случае все равно необходимо, чтобы все имена файлов начинались с подстроки некоторого вида, и это не устранило бы зависимость между сценариями `split` и `join`. Однако поскольку эти сценарии задуманы как два этапа одного и того же процесса, какие-то зависимости между ними выглядят вполне оправданными.

Варианты использования

Проделаем еще несколько экспериментов с этими системными утилитами Python, чтобы продемонстрировать другие режимы работы. Если при запуске в командной строке заданы не все аргументы, сценарии `split` и `join` достаточно сообразительны, чтобы попросить пользователя ввести параметры *интерактивно*. Рассмотрим снова процесс разрезания и склеивания самоустанавливающегося файла Python в Windows, когда параметры вводятся в окне консоли DOS:

```
C:\temp> python C:\...\PP4E\System\Filetools\split.py
File to be split? python-3.1.msi
Directory to store part files? splitout
Splitting C:\temp\python-3.1.msi to C:\temp\splitout by 1433600
Split finished: 10 parts are in C:\temp\splitout
Press Enter key

C:\temp> python C:\...\PP4E\System\Filetools\join.py
Directory containing part files? splitout
Name of file to be recreated? newpy31.msi
Joining C:\temp\splitout to make C:\temp\newpy31.msi
Join complete: see C:\temp\newpy31.msi
Press Enter key

C:\temp> fc /B python-3.1.msi newpy31.msi
Comparing files python-3.1.msi and NEWPY31.MSI
FC: no differences encountered
```

Когда эти программы запускаются в проводнике файлов Windows *двойным щелчком мыши*, они действуют точно так же (при таком способе запуска они обычно не получают аргументов командной строки). В таком режиме вывод абсолютного пути помогает прояснить, где в действительности находятся файлы. Помните, что при запуске щелчком мыши на файле текущим рабочим каталогом является исходный каталог сценария, поэтому при вводе просто имени файла оно будет отнесено в каталог со сценарием. Чтобы поместить файлы фрагментов в другое место, введите полный путь:

```
[во всплывающем окне консоли DOS, когда split.py запущен щелчком мыши]
File to be split? c:\temp\python-3.1.msi
Directory to store part files? c:\temp\parts
Splitting c:\temp\python-3.1.msi to c:\temp\parts by 1433600
Split finished: 10 parts are in c:\temp\parts
Press Enter key

[во всплывающем окне консоли DOS, когда join.py запущен щелчком мыши]
Directory containing part files? c:\temp\parts
Name of file to be recreated? c:\temp\morepy31.msi
Joining c:\temp\parts to make c:\temp\morepy31.msi
Join complete: see c:\temp\morepy31.msi
Press Enter key
```

Поскольку основная логика этих сценариев оформлена в виде функций, их легко можно использовать, *импортировав* и вызвав из другого компонента Python (убедитесь, что путь поиска модулей включает каталог, содержащий корень PP4E, – первая сокращенная строка в следующем примере демонстрирует один из способов добиться этого):

```
C:\temp> set PYTHONPATH=C:\...\dev\Examples
C:\temp> python
>>> from PP4E.System.Filetools.split import split
>>> from PP4E.System.Filetools.join import join
>>>
>>> numparts = split('python-3.1.msi', 'calldir')
>>> numparts
10
>>> join('calldir', 'callpy31.msi')
>>>
>>> import os
>>> os.system('fc /B python-3.1.msi callpy31.msi')
Comparing files python-3.1.msi and CALLPY31.msi
FC: no differences encountered
0
```

Замечание, касающееся производительности: все приведенные здесь примеры запуска сценариев `split` и `join` обрабатывают файл размером 13 Мбайт и выполняются не более 1 секунды реального времени на моем ноутбуке, работающем под управлением Windows 7 и снабженном процессором Atom с тактовой частотой 2 Ггц, – достаточно быстро для любого мыслимого применения. Оба сценария столь же быстро справляются и с другими значениями *размеров фрагментов*. Ниже показано, как выполняется разрезание файла на фрагменты по 4 Мбайта и 500 Кбайт:

```
C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 4000000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 4000000
Split finished: 4 parts are in C:\temp\tempsplit

C:\temp> dir tempsplit
...часть строк опущена...
Directory of C:\temp\tempsplit

02/21/2010  01:27 PM      <DIR>          .
02/21/2010  01:27 PM      <DIR>          ..
02/21/2010  01:27 PM           4,000,000 part0001
02/21/2010  01:27 PM           4,000,000 part0002
02/21/2010  01:27 PM           4,000,000 part0003
02/21/2010  01:27 PM           1,814,272 part0004
               4 File(s)       13,814,272 bytes
               2 Dir(s)    188,671,983,616 bytes free

C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 500000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 500000
```

```
Split finished: 28 parts are in C:\temp\tempsplit

C:\temp> dir tempsplit
...часть строк опущена...
Directory of C:\temp\tempsplit

02/21/2010  01:27 PM      <DIR>          .
02/21/2010  01:27 PM      <DIR>          ..
02/21/2010  01:27 PM             500,000 part0001
02/21/2010  01:27 PM             500,000 part0002
02/21/2010  01:27 PM             500,000 part0003
02/21/2010  01:27 PM             500,000 part0004
02/21/2010  01:27 PM             500,000 part0005
...часть строк опущена...
02/21/2010  01:27 PM             500,000 part0024
02/21/2010  01:27 PM             500,000 part0025
02/21/2010  01:27 PM             500,000 part0026
02/21/2010  01:27 PM             500,000 part0027
02/21/2010  01:27 PM             314,272 part0028
              28 File(s)     13,814,272 bytes
               2 Dir(s)   188,671,946,752 bytes free
```

Разрезание может выполняться заметно дольше, если указать размеры файлов фрагментов настолько маленькими, что это приведет к созданию тысяч фрагментов, – разрезание на 1382 фрагмента выполняется медленнее (впрочем, некоторые современные компьютеры настолько быстрые, что разницу во времени можно не заметить):

```
C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 10000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 10000
Split finished: 1382 parts are in C:\temp\tempsplit

C:\temp> C:\...\PP4E\System\Filetools\join.py tempsplit manypy31.msi
Joining C:\temp\tempsplit to make C:\temp\manypy31.msi
Join complete: see C:\temp\manypy31.msi

C:\temp> fc /B python-3.1.msi manypy31.msi
Comparing files python-3.1.msi and MANYPY31.MSI
FC: no differences encountered

C:\temp> dir tempsplit
...часть строк опущена...
Directory of C:\temp\tempsplit

02/21/2010  01:40 PM      <DIR>          .
02/21/2010  01:40 PM      <DIR>          ..
02/21/2010  01:39 PM              10,000 part0001
02/21/2010  01:39 PM              10,000 part0002
02/21/2010  01:39 PM              10,000 part0003
```

```
02/21/2010  01:39 PM            10,000 part0004
02/21/2010  01:39 PM            10,000 part0005
```

...более 1000 строк опущено...

```
02/21/2010  01:40 PM            10,000 part1378
02/21/2010  01:40 PM            10,000 part1379
02/21/2010  01:40 PM            10,000 part1380
02/21/2010  01:40 PM            10,000 part1381
02/21/2010  01:40 PM             4,272 part1382
            1382 File(s)     13,814,272 bytes
               2 Dir(s)  188,651,008,000 bytes free
```

Наконец, сценарий `split` достаточно умен, чтобы создать выходной каталог, если его не существует, или очистить его от старых файлов, если он существует, – в следующем примере видно, что в выходном каталоге остались только новые файлы. Так как сценарий `join` объединяет все файлы, присутствующие в выходном каталоге, это полезное эргономическое дополнение. Если бы выходной каталог не очищался перед каждым запуском сценария `split`, можно было бы легко забыть, что в каталоге находятся файлы от предыдущего прогона. Поскольку эти сценарии предназначены для выполнения простыми пользователями, они должны быть как можно более снисходительными. Ваши пользователи могут оказаться более подготовленными (хотя вам не следует надеяться на это).

```
C:\temp> C:\...\PP4E\System\Filetools\split.py python-3.1.msi tempsplit 5000000
Splitting C:\temp\python-3.1.msi to C:\temp\tempsplit by 5000000
Split finished: 3 parts are in C:\temp\tempsplit

C:\temp> dir tempsplit
```

...часть строк опущена...

```
Directory of C:\temp\tempsplit

02/21/2010  01:47 PM      <DIR>          .
02/21/2010  01:47 PM      <DIR>          ..
02/21/2010  01:47 PM           5,000,000 part0001
02/21/2010  01:47 PM           5,000,000 part0002
02/21/2010  01:47 PM           3,814,272 part0003
               3 File(s)      13,814,272 bytes
               2 Dir(s)   188,654,452,736 bytes free
```

Конечно, проблему, которую решают эти сценарии, в настоящее время легко можно решить, просто купив более емкую флешку или предоставив детям выход в Интернет. Тем не менее, как только вы решите какую-либо проблему с помощью сценариев, вы поймете, насколько простым и гибким средством является Python, снабженный мощными встроенными инструментами, особенно удобными для разработки сценариев автоматизации, таких как эти. При грамотном использовании Python может стать для вас аналогом швейцарского армейского ножа.

Создание веб–страниц для переадресации

Перемещаться всегда трудно, даже в киберпространстве. Изменение адреса веб-сайта в Интернете может привести ко всякого рода неудобствам. Вам придется просить знакомых использовать новый адрес, а в случае всех остальных – надеяться, что по ходу дела они наткнутся на него сами. Но если ваша работа зависит от Интернета, то перемещение вызывает не меньше проблем, чем смена адреса в реальном мире.

К сожалению, таких перемещений сайта часто невозможно избежать. Как провайдеры интернет-услуг (Internet Service Providers, ISP), 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 веб-адреса.[1]

Представьте себе, однако, что вы пишете книги для издательства O'Reilly и опубликовали адрес своего веб-сайта во многих книгах, продаваемых по всему свету. Что делать, если качество обслуживания вашего провайдера такое, что требуется переместить сайт? Оповещение об этом сотен тысяч читателей не представляется возможным.

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 и обеспечивающую возможность автоматического перемещения через несколько секунд. При наличии таких *файлов со ссылками переадресации* посетители прежнего адреса окажутся на расстоянии одного щелчка от нового адреса.

Звучит довольно просто. Но поскольку посетители могут попытаться непосредственно обратиться по адресу *любого* файла на вашем прежнем сайте, в общей сложности вам потребуется оставить по одному файлу со ссылкой переадресации для каждого прежнего файла – страниц

[1] Это случается. Действительно, почти все, кто проводит в киберпространстве значительное время, могут рассказать пару ужасных историй. Моя состоит в следующем: несколько лет тому назад я пользовался услугами провайдера, который полностью отключился от Интернета на несколько недель из-за проникновения в систему защиты 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 Веб в такой же мере, как мое, вы хорошо представляете, какую панику может вызвать подобное отключение.

HTML, графических файлов и так далее. Если ваш прежний сервер не поддерживает возможность автоматической переадресации (мой не поддерживал), это может превратиться в проблему. 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 содержал более 100 файлов HTML к тому времени, когда я писал этот абзац, перспектива запускать редактор для каждого файла оказалась более чем достаточной мотивацией для выработки автоматизированного решения.

Файл шаблона страницы

Вот что я придумал. Прежде всего, я создал текстовый файл универсального *шаблона страницы*, представленный в примере 6.7 и описывающий вид всех файлов со ссылками переадресации, части которых будут заполнены позднее.

Пример 6.7. PP4E\System\Filetools\template.html

```
<HTML>
<head>
<META HTTP-EQUIV="Refresh" CONTENT="10; URL=http://$server$/$home$/$file$">
<title>Site Redirection Page: $file$</title>
</head>
<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. You will be redirectly shortly.
</P>

<HR>
</BODY></HTML>
```

Чтобы полностью разобраться в этом шаблоне, требуется некоторое знание HTML – языка описания веб-страниц, который мы рассмотрим в четвертой части книги. Но для целей данного примера можно проигнорировать большую часть содержимого этого файла и сосредоточиться только на тех частях, которые окружены знаками доллара: строки `$server$`, `$home$` и `$file$` являются элементами, которые должны быть заменены действительными значениями с помощью операции глобального поиска с заменой. Значения этих элементов зависят от места, куда был перемещен сайт, и от имени файла.

Сценарий генератора страниц

Теперь, имея файл шаблона страницы, сценарий Python из примера 6.8 автоматически сгенерирует все необходимые файлы со ссылками переадресации.

Пример 6.8. PP4E\System\Filetools\site-forward.py

```
"""
###############################################################################
Создает страницы со ссылками переадресации на перемещенный веб-сайт.
Генерирует по одной странице для каждого существующего на сайте файла html;
сгенерированные файлы нужно выгрузить на ваш старый веб-сайт. Смотрите описание
ftplib далее в книге, где представлены приемы реализации выгрузки файлов
в сценариях после или в процессе создания файлов страниц.
###############################################################################
"""
import os
servername = 'learning-python.com'    # новый адрес сайта
homedir = 'books'                     # корневой каталог сайта
sitefilesdir = r'C:\temp\public_html' # локальный каталог с файлами сайта
uploaddir = r'C:\temp\isp-forward'    # где сохранять сгенерированные файлы
templatename = 'template.html'        # шаблон для генерируемых страниц

try:
    os.mkdir(uploaddir)               # при необходимости создать каталог для
except OSError: pass                  # выгружаемых страниц

template = open(templatename).read()  # загрузить или импортировать шаблон
sitefiles = os.listdir(sitefilesdir)  # имена файлов без пути к ним

count = 0
for filename in sitefiles:
    if filename.endswith('.html') or filename.endswith('.htm'):
        fwdname = os.path.join(uploaddir, filename)
        print('creating', filename, 'as', fwdname)
        filetext = template.replace('$server$', servername) # вставить текст
        filetext = filetext.replace('$home$', homedir)      # и записать
        filetext = filetext.replace('$file$', filename)     # измененный файл
        open(fwdname, 'w').write(filetext)
        count += 1

print('Last file =>\n', filetext, sep='')
print('Done:', count, 'forward files created.')
```

Обратите внимание, что текст шаблона загружается путем чтения *файла*, – шаблон можно было бы реализовать в виде импортируемой строковой переменной Python (например, в виде строки в тройных кавычках в файле модуля). Заметьте также, что все параметры конфигурации задаются инструкциями присваивания в начале сценария, а не аргумен-

тами 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 глобальную подстановку с различными именами файлов для каждого генерируемого файла. Фактически файл шаблона можно изменить как угодно, и это никак не отразится на сценарии. Такое разделение труда может быть использовано в самых разных ситуациях – при создании «make-файлов», бланков писем, ответов CGI-сценариев на веб-сервере и так далее. Что касается библиотечных инструментов, сценарий генератора:

- Использует функцию `os.listdir` для обхода всех имен файлов в каталоге сайта (точно так же можно было бы использовать функцию `glob.glob`, но при этом потребовалось бы удалять пути к файлам из их имен)
- Использует строковый метод `replace` для поиска и замены элементов в тексте файла шаблона, ограниченных символами `$`, и метод `endswith`, чтобы пропустить файлы, не являющиеся страницами HTML (например, изображения – большинство броузеров не знают, что делать с разметкой HTML в файлах «.jpg»)
- Использует функцию `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 его выполнения практически на любой платформе Python. Я запускал его на ноутбуке, работающем под управлением Windows (на котором я пишу эту книгу), а также на Linux-сервере (где находится мой сайт *http://learning-python.com*). Ниже показан пример запуска этого сценария в Windows:

```
C:\...\PP4E\System\Filetools> python site-forward.py
creating about-lp.html as C:\temp\isp-forward\about-lp.html
creating about-lp1e.html as C:\temp\isp-forward\about-lp1e.html
creating about-lp2e.html as C:\temp\isp-forward\about-lp2e.html
creating about-lp3e.html as C:\temp\isp-forward\about-lp3e.html
creating about-lp4e.html as C:\temp\isp-forward\about-lp4e.html
...множество строк удалено...
creating training.html as C:\temp\isp-forward\training.html
creating whatsnew.html as C:\temp\isp-forward\whatsnew.html
creating whatsold.html as C:\temp\isp-forward\whatsold.html
creating xlate-lp.html as C:\temp\isp-forward\xlate-lp.html
creating zopeoutline.htm as C:\temp\isp-forward\zopeoutline.htm
Last file =>
```

```
<HTML>
<head>
<META HTTP-EQUIV="Refresh" CONTENT="10; URL=http://learning-python.com/books/zop
eoutline.htm">
<title>Site Redirection Page: zopeoutline.htm</title>
</head>
<BODY>

<H1>This page has moved</H1>
<P>This page now lives at this address:

<P><A HREF="http://learning-python.com/books/zopeoutline.htm">
http://learning-python.com/books/zopeoutline.htm</A>

<P>Please click on the new address to jump to this page, and
update any links accordingly. You will be redirectly shortly.
</P>

<HR>
</BODY></HTML>
Done: 124 forward files created.
```

Чтобы проверить результаты работы сценария, щелкните дважды на любом сгенерированном файле и посмотрите, как он выглядит в веб-броузере (или выполните команду `start` в окне консоли DOS, например `start isp-forward\about-lp4e.html`). На рис. 6.1 показано, как выглядит одна из сгенерированных страниц на моем компьютере.

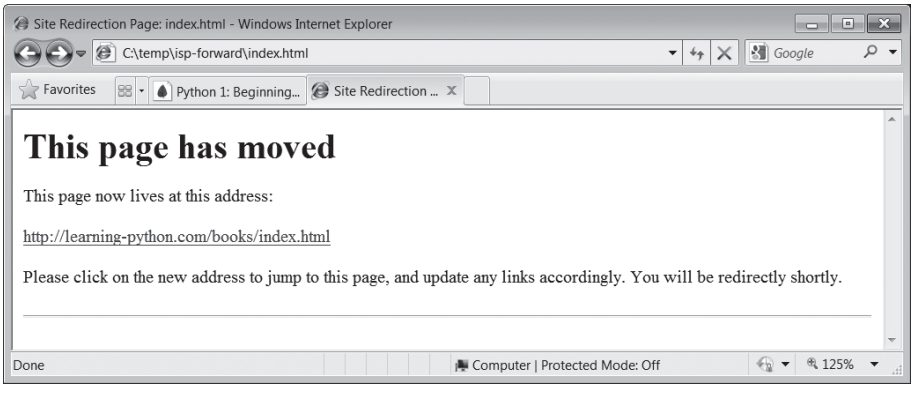


Рис. 6.1. *Сгенерированная страница переадресации сайта*

Для завершения процесса еще необходимо установить ссылки переадресации: выгрузите 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 13 (эту работу выполняет сценарий *PP4E\Internet\Ftp*

uploadflat.py). Всерьез занявшись написанием сценариев, вы поразитесь тому, какой объем ручного труда можно автоматизировать с помощью Python. В следующем разделе представлен еще один большой пример.

Сценарий регрессивного тестирования

Рано или поздно ошибки случаются. Как мы уже видели, Python предоставляет интерфейсы доступа к различным системным службам, а также инструменты для добавления новых интерфейсов. В примере 6.9 показаны некоторые часто используемые системные инструменты в действии. В нем реализована простая система *регрессивного тестирования* сценариев на языке Python. Она запускает каждый сценарий Python в указанном каталоге с заданным набором входных файлов и аргументов командной строки и сравнивает вывод каждого прогона с предыдущими результатами. Этот сценарий можно было бы использовать в качестве автоматизированной системы тестирования, 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 6.9. PP4E\System\Tester\tester.py

```
"""
################################################################################
Тестирует сценарии Python в каталоге, передает им аргументы командной строки,
выполняет перенаправление stdin, перехватывает stdout, stderr и код завершения,
чтобы определить наличие ошибок и отклонений от предыдущих результатов
выполнения. Запуск сценариев и управление потоками ввода-вывода производится
с помощью переносимого модуля subprocess (как это делает функция os.popen3
в Python 2.X). Потоки ввода-вывода всегда интерпретируются модулем subprocess
как двоичные. Стандартный ввод, аргументы, стандартный вывод и стандартный вывод
ошибок отображаются в файлы, находящиеся в подкаталогах.

Этот сценарий командной строки позволяет указать имя тестируемого каталога
и флаг принудительной генерации выходного файла. Этот программный код можно было
бы упаковать в функцию, однако то обстоятельство, что результатами сценария
являются сообщения и выходные файлы, снижает практическую пользу модели вызов/
возвращаемое значение.

Дополнительные возможные расширения: можно было бы реализовать по несколько
наборов аргументов командной строки и/или входных файлов для каждого
тестируемого сценария и запускать их по несколько раз (использовать функцию glob
для выборки нескольких файлов ".in*" в каталоге Inputs).
Возможно, было бы проще хранить все файлы, необходимые для проведения тестов,
в одном и том же каталоге, но с различными расширениями, однако с течением
времени их объем мог бы оказаться слишком большим.
В случае ошибок можно было бы сохранять содержимое потоков вывода stderr
и stdout в подкаталоге Errors, но я предпочитаю иметь ожидаемый/фактический
вывод в подкаталоге Outputs.
```

```
###############################################################################
"""
import os, sys, glob, time
from subprocess import Popen, PIPE

# конфигурационные аргументы
testdir = sys.argv[1] if len(sys.argv) > 1 else os.curdir
forcegen = len(sys.argv) > 2
print('Start tester:', time.asctime())
print('in', os.path.abspath(testdir))

def verbose(*args):
    print('-'*80)
    for arg in args: print(arg)
def quiet(*args): pass
trace = quiet

# отбор сценариев для тестирования
testpatt = os.path.join(testdir, 'Scripts', '*.py')
testfiles = glob.glob(testpatt)
testfiles.sort()
trace(os.getcwd(), *testfiles)

numfail = 0
for testpath in testfiles:                # протестировать все сценарии
    testname = os.path.basename(testpath) # отбросить путь к файлу

    # получить входной файл и аргументы для тестируемого сценария
    infile = testname.replace('.py', '.in')
    inpath = os.path.join(testdir, 'Inputs', infile)
    indata = open(inpath, 'rb').read() if os.path.exists(inpath) else b''

    argfile = testname.replace('.py', '.args')
    argpath = os.path.join(testdir, 'Args', argfile)
    argdata = open(argpath).read() if os.path.exists(argpath) else ''

    # местоположение файлов для сохранения stdout и stderr,
    # очистить предыдущие результаты
    outfile = testname.replace('.py', '.out')
    outpath = os.path.join(testdir, 'Outputs', outfile)
    outpathbad = outpath + '.bad'
    if os.path.exists(outpathbad): os.remove(outpathbad)

    errfile = testname.replace('.py', '.err')
    errpath = os.path.join(testdir, 'Errors', errfile)
    if os.path.exists(errpath): os.remove(errpath)

    # запустить тестируемый сценарий, перенаправив потоки ввода-вывода
    pypath = sys.executable
    command = '%s %s %s' % (pypath, testpath, argdata)
    trace(command, indata)
```

```
        process = Popen(command, shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
        process.stdin.write(indata)
        process.stdin.close()
        outdata = process.stdout.read()
        errdata = process.stderr.read()    # при работе с двоичными файлами
        exitstatus = process.wait()        # данные имеют тип bytes
        trace(outdata, errdata, exitstatus)

        # проанализировать результаты
        if exitstatus != 0:
            print('ERROR status:', testname, exitstatus) # код заверш.
        if errdata:                                      # и/или stderr
            print('ERROR stream:', testname, errpath)    # сохр. текст ошибки
            open(errpath, 'wb').write(errdata)

        if exitstatus or errdata:                        # оба признака ошибки
            numfail += 1              # можно получить код завершения + код ошибки
            open(outpathbad, 'wb').write(outdata)        # сохранить вывод

        elif not os.path.exists(outpath) or forcegen:
            print('generating:', outpath)                # создать файл, если
            open(outpath, 'wb').write(outdata)           # необходимо

        else:
            priorout = open(outpath, 'rb').read()        # или сравнить с прежними
                                                         # результатами
            if priorout == outdata:
                print('passed:', testname)
            else:
                numfail += 1
                print('FAILED output:', testname, outpathbad)
                open(outpathbad, 'wb').write(outdata)

    print('Finished:', time.asctime())
    print('%s tests were run, %s tests failed.' % (len(testfiles), numfail))
```

Мы уже познакомились с инструментами, используемыми этим сценарием, выше в этой части книги – с модулем `subprocess`, с функциями `os.path`, `glob`, с файлами и другими. Этот пример в значительной степени просто объединяет эти инструменты для решения поставленной задачи. Основной операцией в сценарии является сравнение нового и старого вывода с целью обнаружить изменения («регрессии»). Попутно он также манипулирует аргументами командной строки, сообщениями об ошибках, кодами завершения и файлами.

Кроме того, этот сценарий не только самый большой из тех, что встречались нам до сих пор; это также самый практичный и представительный инструмент системного администрирования (он был разработан на основе похожего инструмента, который я использовал в прошлом для выявления изменений в компиляторе). Самый лучший способ понять, как он действует, вероятно, состоит в том, чтобы продемонстрировать его на

конкретных примерах. В следующем разделе демонстрируется сеанс тестирования и попутно даются пояснения по реализации сценария.

Запускаем тестирование

Основная магия сценария, выполняющего тестирование, представленного в примере 6.9, заключена в используемой им структуре каталогов. При первом запуске в каталоге тестирования (или если вы заставляете его начать сначала, передавая ему второй аргумент командной строки) сценарий:

- Составит список тестируемых сценариев в подкаталоге `Scripts`
- Извлечет ассоциированные с тестируемым сценарием входной файл и аргументы командной строки из подкаталогов `Inputs` и `Args`
- Сгенерирует начальные файлы для стандартного потока вывода *stdout*, которые обычно помещаются в подкаталог `Outputs`
- Сообщит о сценариях, в процессе тестирования которых либо появились сообщения об ошибках в потоке *stderr*, либо код завершения отличается от нуля

В случае любых ошибок, обнаруженных при тестировании сценария, сохраняется содержимое потока *stderr* с текстом сообщений об ошибках, а также полный вывод, сгенерированный до момента ошибки. Текст из стандартного потока ошибок сохраняется в файл в подкаталоге `Errors`. Содержимое стандартного вывода в случае обнаружения ошибок сохраняется в файл, имя которого имеет специальное расширение «.bad» в подкаталоге `Outputs` (сохранение в файле с нормальным именем в подкаталоге `Outputs` привело бы к ошибке тестирования после устранения ошибки в тестируемом сценарии!). Ниже приводится пример первого запуска:

```
C:\...\PP4E\System\Tester> python tester.py . 1
Start tester: Mon Feb 22 22:13:38 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
generating: .\Outputs\test-basic-args.out
generating: .\Outputs\test-basic-stdout.out
generating: .\Outputs\test-basic-streams.out
generating: .\Outputs\test-basic-this.out
ERROR status: test-errors-runtime.py 1
ERROR stream: test-errors-runtime.py .\Errors\test-errors-runtime.err
ERROR status: test-errors-syntax.py 1
ERROR stream: test-errors-syntax.py .\Errors\test-errors-syntax.err
ERROR status: test-status-bad.py 42
generating: .\Outputs\test-status-good.out
Finished: Mon Feb 22 22:13:41 2010
8 tests were run, 3 tests failed.
```

При запуске каждого сценария тестирующий сценарий устанавливает все заданные аргументы командной строки, передает все входные данные (если таковые имеются) в канал ввода и перехватывает стандарт-

ные потоки вывода и ошибок, сохраняет код завершения. Когда я запускал этот пример, в каталоге находилось 8 тестируемых сценариев, а также множество входных и выходных файлов. Поскольку схема именования каталогов и файлов имеет ключевое значение для данного примера, ниже приводится список с содержимым тестового каталога для этого сеанса – основным является каталог `Scripts`, потому что в нем собраны тестируемые сценарии:

```
C:\...\PP4E\System\Tester> dir /B
Args
Errors
Inputs
Outputs
Scripts
tester.py
xxold

C:\...\PP4E\System\Tester> dir /B Scripts
test-basic-args.py
test-basic-stdout.py
test-basic-streams.py
test-basic-this.py
test-errors-runtime.py
test-errors-syntax.py
test-status-bad.py
test-status-good.py
```

Другие подкаталоги содержат все необходимые входные данные и выходные файлы с результатами работы тестируемых сценариев:

```
C:\...\PP4E\System\Tester> dir /B Args
test-basic-args.args
test-status-good.args

C:\...\PP4E\System\Tester> dir /B Inputs
test-basic-args.in
test-basic-streams.in

C:\...\PP4E\System\Tester> dir /B Outputs
test-basic-args.out
test-basic-stdout.out
test-basic-streams.out
test-basic-this.out
test-errors-runtime.out.bad
test-errors-syntax.out.bad
test-status-bad.out.bad
test-status-good.out

C:\...\PP4E\System\Tester> dir /B Errors
test-errors-runtime.err
test-errors-syntax.err
```

Я не буду приводить здесь содержимое всех файлов (как видите, их достаточно много и все они входят в состав пакета с примерами для данной книги), но, чтобы вы могли получить некоторое представление, ниже приводится содержимое файлов, ассоциированных с тестируемым сценарием *test-basic-args.py*:

```
C:\...\PP4E\System\Tester> type Scripts\test-basic-args.py
# аргументы, потоки
import sys, os
print(os.getcwd())                    # в Outputs
print(sys.path[0])

print('[argv]')
for arg in sys.argv:                  # из Args
    print(arg)                        # в Outputs

print('[interaction]')                # в Outputs
text = input('Enter text:')           # из Inputs
rept = sys.stdin.readline()           # из Inputs
sys.stdout.write(text * int(rept))    # в Outputs

C:\...\PP4E\System\Tester> type Args\test-basic-args.args
-command -line --stuff

C:\...\PP4E\System\Tester> type Inputs\test-basic-args.in
Eggs
10

C:\...\PP4E\System\Tester> type Outputs\test-basic-args.out
C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester\Scripts
[argv]
.\Scripts\test-basic-args.py
-command
-line
--stuff
[interaction]
Enter text:EggsEggsEggsEggsEggsEggsEggsEggsEggsEggs
```

А еще ниже – два файла, связанные с одной из обнаруженных ошибок. Первый из них хранит содержимое потока *stderr*, а второй – содержимое потока *stdout*, сгенерированное до момента появления ошибки. Они предназначены для анализа человеком (или с помощью других инструментов) и автоматически будут удалены в следующем сеансе тестирования:

```
C:\...\PP4E\System\Tester> type Errors\test-errors-runtime.err
Traceback (most recent call last):
  File ".\Scripts\test-errors-runtime.py", line 3, in <module>
    print(1 / 0)
ZeroDivisionError: int division or modulo by zero
```

```
C:\...\PP4E\System\Tester> type Outputs\test-errors-runtime.out.bad
starting
```

Если теперь запустить тестирование еще раз, не внося никаких изменений в тестируемые сценарии, тестирующий сценарий сравнит сохраненные ранее результаты с новыми и определит отсутствие регрессий. Об ошибках, которые определяются по коду завершения, и о наличии сообщений в потоке *stderr* будет сообщаться, как и прежде, но в других тестах не будет обнаружено никаких отклонений от сохраненных ранее результатов:

```
C:\...\PP4E\System\Tester> python tester.py
Start tester: Mon Feb 22 22:26:41 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
passed: test-basic-args.py
passed: test-basic-stdout.py
passed: test-basic-streams.py
passed: test-basic-this.py
ERROR status: test-errors-runtime.py 1
ERROR stream: test-errors-runtime.py .\Errors\test-errors-runtime.err
ERROR status: test-errors-syntax.py 1
ERROR stream: test-errors-syntax.py .\Errors\test-errors-syntax.err
ERROR status: test-status-bad.py 42
passed: test-status-good.py
Finished: Mon Feb 22 22:26:43 2010
8 tests were run, 3 tests failed.
```

Но когда я внес в один из тестируемых сценариев изменение, повлиявшее на его вывод (я изменил счетчик цикла, чтобы сценарий выводил меньше строк), тестирующий сценарий обнаружил регрессию и сообщил о ней – отличия между новым и старым выводом были восприняты как ошибка прохождения теста, и новый вывод был сохранен в подкаталоге `Outputs` в файле с расширением «.bad»:

```
C:\...\PP4E\System\Tester> python tester.py
Start tester: Mon Feb 22 22:28:35 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
passed: test-basic-args.py
FAILED output: test-basic-stdout.py .\Outputs\test-basic-stdout.out.bad
passed: test-basic-streams.py
passed: test-basic-this.py
ERROR status: test-errors-runtime.py 1
ERROR stream: test-errors-runtime.py .\Errors\test-errors-runtime.err
ERROR status: test-errors-syntax.py 1
ERROR stream: test-errors-syntax.py .\Errors\test-errors-syntax.err
ERROR status: test-status-bad.py 42
passed: test-status-good.py
Finished: Mon Feb 22 22:28:38 2010
8 tests were run, 4 tests failed.

C:\...\PP4E\System\Tester> type Outputs\test-basic-stdout.out.bad
begin
```

```
Spam!
Spam! Spam!
Spam! Spam! Spam!
Spam! Spam! Spam! Spam!
end
```

И еще одно замечание по использованию: если переменную `trace` в этом сценарии установить в значение `verbose`, он будет выводить более подробные сообщения, которые помогут вам проследить за порядком выполнения программы (но, вероятно, чересчур подробные для практического применения):

```
C:\...\PP4E\System\Tester> tester.py
Start tester: Mon Feb 22 22:34:51 2010
in C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
--------------------------------------------------------------------------------
C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Tester
.\Scripts\test-basic-args.py
.\Scripts\test-basic-stdout.py
.\Scripts\test-basic-streams.py
.\Scripts\test-basic-this.py
.\Scripts\test-errors-runtime.py
.\Scripts\test-errors-syntax.py
.\Scripts\test-status-bad.py
.\Scripts\test-status-good.py
--------------------------------------------------------------------------------
C:\Python31\python.exe .\Scripts\test-basic-args.py -command -line --stuff
b'Eggs\r\n10\r\n'
--------------------------------------------------------------------------------
b'C:\\Users\\mark\\Stuff\\Books\\4E\\PP4E\\dev\\Examples\\PP4E\\System\\Tester\r
\nC:\\Users\\mark\\Stuff\\Books\\4E\\PP4E\\dev\\Examples\\PP4E\\System\\Tester\\
Scripts\r\n[argv]\r\n.\\Scripts\\test-basic-args.py\r\n-command\r\n-line\r\n--st
uff\r\n[interaction]\r\nEnter text:EggsEggsEggsEggsEggsEggsEggsEggsEggs'
b''
0
passed: test-basic-args.py
```

...множество строк удалено...

Изучите внимательнее реализацию тестирующего сценария, чтобы получить о нем более полное представление. Естественно, о тестировании как таковом можно было бы рассказать намного больше, чем позволяет пространство книги. Например, для тестирования сценариев необязательно запускать их в дочерних процессах и вполне можно обойтись импортированием модулей и тестированием с помощью обработчиков исключений в инструкциях `try`. Кроме того, наш тестирующий сценарий можно было бы расширять и совершенствовать в самых разных направлениях (некоторые предложения приводятся в строке документирования). Более того, в состав Python входят два фреймворка тестирования, `doctest` и `unittest` (он же PyUnit), которые предоставляют инструменты и структуры для создания регрессивных и модульных тестов:

unittest

Объектно-ориентированный фреймворк, который позволяет определять совокупности тестовых данных, ожидаемые результаты и комплекты тестов. Подклассы предоставляют методы тестирования и используют унаследованные методы проверки для определения ожидаемых результатов.

doctest

Анализирует и выполняет тесты, представленные в виде листингов интерактивного сеанса в строках документирования внутри тестируемого модуля. В листингах определяются тестовые вызовы и ожидаемые результаты – фреймворк doctest по сути повторно выполняет интерактивный сеанс.

За дополнительной информацией об инструментах и способах тестирования, сторонних или входящих в состав Python, обращайтесь к руководству по библиотеке Python, на веб-сайт PyPI и к своим любимым поисковым системам.

Тем не менее наш сценарий прекрасно справляется с задачей автоматизации тестирования сценариев командной строки на языке Python, которые выполняются как независимые программы в стандартном контексте выполнения. Поскольку наш сценарий полностью независим от тестируемых сценариев, мы можем определять новые совокупности тестовых данных без необходимости изменять реализацию тестирующего сценария. А так как он написан на языке Python, его легко и быстро можно приспосабливать под новые потребности. Как мы увидим еще раз в следующем разделе, такая простота, обеспечиваемая языком Python, может оказаться решающим преимуществом для реальных задач.

Тест прошел неудачно?

Когда в главе 13 мы узнаем, как из сценариев на языке Python отправлять электронную почту, вы, возможно, захотите улучшить этот сценарий так, чтобы он автоматически отправлял письмо в случае неудачи регулярно выполняемого теста (например, с помощью планировщика заданий `cron` в Unix). Благодаря этому не нужно будет даже помнить о необходимости проверить результаты. Конечно, можно развивать его еще дальше.

В одной компании я работал над добавлением звуковых эффектов в компилятор тестовых сценариев: вы слышали аплодисменты, если в процессе тестирования не было обнаружено регрессий, и совсем другие звуки в противном случае. (Советы по воспроизведению звуков смотрите в конце данной главы, в файле *playfile.py*.)

> В другой компании, где я раньше работал, каждый вечер выполнялся тестирующий сценарий, который автоматически предотвращал сохранение файла с исходным программным кодом, вызвавшим регрессию в тесте, и посылал гневное письмо виновной стороне (и их руководству). Никто не уйдет от испанской инквизиции!

Копирование деревьев каталогов

Мой пишущий привод компакт-дисков иногда делает странные вещи. Копии файлов с необычными именами могут оказаться совершенно испорченными на CD, несмотря на то, что остальные файлы оказываются целыми и невредимыми. При этом не обязательно идет насмарку вся работа – если в большой резервной копии на CD испорчено лишь несколько файлов, я всегда могу поштучно скопировать проблемные файлы на дискеты. К несчастью, копирование в Windows путем перетаскивания мышью 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.

Это ограничение касается не только копий на компакт-дисках. Я столкнулся с подобной проблемой, когда попытался создать резервную копию жесткого диска моего ноутбука на другом жестком диске, – операция копирования, инициированная перетаскиванием файлов мышью, прекращалась, как только встречался файл со слишком длинным или необычным именем (обычно среди сохраненных веб-страниц). Последние полчаса, потраченные на копирование, оказывались потраченными впустую – я был, мягко говоря, расстроен!

Возможно, в Windows существует какая-нибудь волшебная настройка, позволяющая справиться с этой особенностью, но я перестал ее разыскивать, когда понял, что проще написать сценарий копирования на языке Python. Сценарий *cpall.py*, представленный в примере 6.10, реализует один из возможных способов копирования. С его помощью я могу управлять действиями, которые выполняются при обнаружении проблемных файлов, например, пропустить файл с помощью обработчика исключения. Кроме того, этот инструмент работает и на других платформах с тем же интерфейсом и таким же результатом. По крайней мере, мне кажется, что потратить несколько минут и написать на языке Python переносимый и многократно используемый сценарий для решения некоторой задачи более выгодно, чем искать решения, работающие только на одной платформе (если они вообще есть).

Пример 6.10. PP4E\System\Filetools\cpall.py

```
"""
###############################################################################
Порядок использования: "python cpall.py dirFrom dirTo".
Рекурсивно копирует дерево каталогов. Действует подобно команде Unix "cp -r
dirFrom/* dirTo", предполагая, что оба аргумента dirFrom и dirTo являются
именами каталогов.
Был написан с целью обойти фатальные ошибки при копировании файлов
перетаскиванием мышью в Windows (когда встреча первого же проблемного файла
вызывает прекращение операции копирования) и обеспечить возможность реализации
более специализированных операций копирования на языке Python.
###############################################################################
"""
import os, sys
maxfileload = 1000000
blksize = 1024 * 500

def copyfile(pathFrom, pathTo, maxfileload=maxfileload):
    """
    Копирует один файл из pathFrom в pathTo, байт в байт;
    использует двоичный режим для подавления операций
    кодирования/декодирования и преобразований символов конца строки
    """
    if os.path.getsize(pathFrom) <= maxfileload:
        bytesFrom = open(pathFrom,'rb').read() # маленький файл читать целиком
        open(pathTo, 'wb').write(bytesFrom)
    else:
        fileFrom = open(pathFrom, 'rb')        # большие файлы - по частям
        fileTo = open(pathTo, 'wb')            # режим b для обоих файлов
        while True:
            bytesFrom = fileFrom.read(blksize) # прочитать очередной блок
            if not bytesFrom: break            # пустой после последнего блока
            fileTo.write(bytesFrom)

def copytree(dirFrom, dirTo, verbose=0):
    """
    Копирует содержимое dirFrom и вложенных подкаталогов в dirTo,
    возвращает счетчики (files, dirs);
    для представления имен каталогов, недекодируемых на других платформах,
    может потребоваться использовать переменные типа bytes;
    в Unix может потребоваться выполнять дополнительные проверки типов файлов,
    чтобы пропускать ссылки, файлы fifo и так далее.
    """
    fcount = dcount = 0
    for filename in os.listdir(dirFrom):       # для файлов/каталогов
        pathFrom = os.path.join(dirFrom, filename)
        pathTo = os.path.join(dirTo, filename) # расширить оба пути
        if not os.path.isdir(pathFrom):        # скопировать простые файлы
            try:
                if verbose > 1: print('copying', pathFrom, 'to', pathTo)
                copyfile(pathFrom, pathTo)
```

```
                fcount += 1
            except:
                print('Error copying', pathFrom, 'to', pathTo, '--skipped')
                print(sys.exc_info()[0], sys.exc_info()[1])
            else:
                if verbose: print('copying dir', pathFrom, 'to', pathTo)
                try:
                    os.mkdir(pathTo)           # создать новый подкаталог
                    below = copytree(pathFrom, pathTo) # спуск в подкаталоги
                    fcount += below[0]         # увеличить счетчики
                    dcount += below[1]         # подкаталогов
                    dcount += 1
                except:
                    print('Error creating', pathTo, '--skipped')
                    print(sys.exc_info()[0], sys.exc_info()[1])
    return (fcount, dcount)

def getargs():
    """
    Извлекает и проверяет аргументы с именами каталогов, по умолчанию
    возвращает None в случае ошибки
    """
    try:
        dirFrom, dirTo = sys.argv[1:]
    except:
        print('Usage error: 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 hasattr(os.path, 'samefile'):
                same = os.path.samefile(dirFrom, dirTo)
            else:
                same = os.path.abspath(dirFrom) == os.path.abspath(dirTo)
            if same:
                print('Error: dirFrom same as dirTo')
            else:
                return (dirFrom, dirTo)

if __name__ == '__main__':
    import time
    dirstuple = getargs()
    if dirstuple:
        print('Copying...')
        start = time.clock()
        fcount, dcount = copytree(*dirstuple)
```

```
print('Copied', fcount, 'files,', dcount, 'directories', end=' ')
print('in', time.clock() - start, 'seconds')
```

В этом сценарии реализована собственная логика рекурсивного обхода дерева каталогов, в ходе которого запоминаются пути каталогов источника и приемника. На каждом уровне она копирует простые файлы, создает каталоги в целевом пути и производит рекурсивный спуск в подкаталоги с расширением путей «из» и «в» на один уровень. Эту задачу можно запрограммировать и другими способами (например, в процессе обхода можно изменять текущий рабочий каталог с помощью функции `os.chdir` или использовать решение на основе функции `os.walk`, замещая пути «из» и «в» по мере их обхода), но на практике вполне достаточно использовать прием расширения имен каталогов при спуске.

Обратите внимание на повторно используемую в этом сценарии функцию `copyfile` – на тот случай, если потребуется копировать файлы размером в несколько гигабайтов, она, исходя из размера файла, решает, читать ли файл целиком или по частям (напомню, что при вызове без аргументов метода `read` файла он загружает весь файл в строку, находящуюся в памяти). Мы выбрали достаточно большие размеры для читаемых целиком файлов и для блоков, потому что чем больший объем мы будем читать за один присест, тем быстрее будет работать сценарий. Это решение гораздо эффективнее, чем могло бы показаться на первый взгляд, – строки, остающиеся в памяти после последней операции чтения, будут утилизироваться сборщиком мусора, и освободившаяся память будет повторно использована последующими операциями. Здесь мы снова используем двоичный режим доступа к файлам, чтобы подавить кодирование/декодирование содержимого файлов и преобразование символов конца строки – в дереве каталогов могут находиться файлы самых разных типов.

Заметьте также, что сценарий при необходимости создает целевой каталог, и перед началом копирования предполагает, что он пуст, – удалите целевой каталог перед копированием нового дерева с тем же именем, иначе к дереву результата могут присоединиться старые файлы (мы могли бы автоматически очищать целевой каталог перед копированием, но это не всегда бывает желательно). Кроме того, данный сценарий пытается определить – не являются ли исходный и конечный каталоги одним и тем же каталогом. В Unix-подобных системах, где есть такие диковины, как ссылки, функция `os.path.samefile` проделывает более сложную работу, чем простое сравнение абсолютных имен файлов (разные имена файлов могут означать один и тот же файл).

Ниже приводится пример копирования большого дерева примеров книги в Windows (на протяжении этой главы я буду использовать дерево с примерами к предыдущему изданию). При запуске сценарию необходимо указать имена исходного и целевого каталогов, перенаправить вывод сценария в файл, если возникает слишком много ошибок, чтобы можно было прочитать все сообщения о них сразу (например, `> output.`

txt), и при необходимости выполнить команду оболочки rm -r или rmdir /S (или аналогичную для соответствующей платформы), чтобы сначала удалить целевой каталог:

```
C:\...\PP4E\System\Filetools> rmdir /S copytemp
copytemp, Are you sure (Y/N)? y

C:\...\PP4E\System\Filetools> cpall.py C:\temp\PP3E\Examples copytemp
Note: dirTo was created
Copying...
Copied 1430 files, 185 directories in 10.4470980971 seconds

C:\...\PP4E\System\Filetools> fc /B copytemp\PP3E\Launcher.py
                                  C:\temp\PP3E\Examples\PP3E\Launcher.py
Comparing files COPYTEMP\PP3E\Launcher.py and C:\TEMP\PP3E\EXAMPLES\PP3E\
LAUNCHER.PY
FC: no differences encountered
```

Вы можете воспользоваться аргументом verbose функции копирования, чтобы проследить, как протекает процесс копирования. Когда я работал над этим изданием в 2010 году, в этом примере за 10 секунд было скопировано дерево каталогов, содержащее 1430 файлов и 185 подкаталогов, – на моем удручающе медлительном нетбуке (для получения системного времени была использована встроенная функция time.clock). У вас аналогичная операция может выполняться быстрее или медленнее. Во всяком случае, это не хуже, чем самые лучшие результаты хронометража, полученные мной при перетаскивании каталогов мышью на этом же компьютере.

Каким же образом этот сценарий справляется с проблемными файлами на CD с резервными копиями? Секрет заключается в том, что он перехватывает и игнорирует *исключения* и продолжает обход. Чтобы скопировать с CD все хорошие файлы, я просто выполняю команду такого вида:

```
C:\...\PP4E\System\Filetools> python cpall.py G:\Examples C:\PP3E\Examples
```

Поскольку на моем компьютере, работающем под управлением Windows, привод CD доступен как диск «G:», эта команда эквивалентна копированию путем перетаскивания элемента, находящегося в папке верхнего уровня на компакт-диске, за исключением того, что сценарий Python восстанавливается после возникающих ошибок и копирует остальные файлы. В случае ошибки копирования он выводит сообщение в стандартный поток вывода и продолжает работу. При копировании большого количества файлов, вероятно, будет удобнее перенаправить стандартный вывод сценария в файл, чтобы позднее его можно было детально исследовать.

Вообще говоря, сценарию cpall можно передать любой абсолютный путь к каталогу на вашем компьютере, даже такой, который обозначает устройство, например привод CD. Для выполнения сценария в Linux

можно обратиться к приводу CD, указав такой каталог, как */dev/cdrom*. После копирования дерева каталогов таким способом у вас может появиться желание проверить получившийся результат. Чтобы увидеть, как это делается, перейдем к следующему примеру.

Сравнение деревьев каталогов

Инженеры могут иметь параноидальные наклонности (но я вам этого не говорил). По крайней мере, ко мне это относится. Я полагаю, что все это от многолетнего наблюдения за встречающимися ошибками. Например, когда я создаю CD с резервной копией жесткого диска, то ощущаю, что за этим процессом стоит нечто слишком таинственное, чтобы я мог положиться на то, что делает программа записи CD. Может быть, я неправ, но я не могу безоговорочно доверять инструментам, которые временами калечат файлы и, кажется, выбрали третий четверг каждого месяца, чтобы вызывать крах моего компьютера с Windows. В общем, в решающий момент неплохо было бы иметь возможность как можно скорее убедиться, что скопированные на CD данные совпадают с исходными, или хотя бы найти отклонения. Если резервная копия вообще нужна, то такая возможность когда-нибудь *действительно* понадобится.

Поскольку доступ к компакт-дискам с данными осуществляется как к обычным деревьям каталогов, мы снова попадаем в сферу действия инструментов обхода деревьев – для проверки CD с резервной копией достаточно просто выполнить обход его каталога верхнего уровня. Если написать достаточно универсальный сценарий, его также можно будет использовать для проверки других операций копирования, например загруженных tar-файлов, резервных копий жестких дисков и так далее. Фактически, объединив сценарий `cpall` из предыдущего раздела с универсальным инструментом сравнения деревьев, мы получили бы переносимый и легко настраиваемый способ копирования и проверки наборов данных.

Мы уже познакомились с некоторыми механизмами обхода деревьев каталогов, но мы не сможем здесь воспользоваться ими непосредственно: нам необходимо параллельно обойти *два* каталога и попутно исследовать встречающиеся файлы. Кроме того, обход только одного из двух каталогов не позволит выявить файлы и каталоги, существующие только в другом каталоге. Здесь требуется некоторое более специализированное и рекурсивное решение.

Поиск расхождений между каталогами

Прежде чем приступать к программированию, 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 в общем случае деревья могут иметь произвольную различную форму, глубину и так далее.

В более общем случае каталог в одном дереве может содержать больше или меньше элементов, чем соответствующий каталог в другом дереве. Если различие обусловлено наличием других файлов, это означает отсутствие соответствующих файлов для сравнения в другом каталоге. Если различие обусловлено наличием других каталогов, это означает отсутствие соответствующей ветви, в которую нужно войти. На самом деле единственный способ выявить файлы и каталоги, которые есть в одном дереве, но отсутствуют в другом, заключается в том, чтобы выявить различия в каталогах каждого уровня.

Иными словами, алгоритм сравнения деревьев должен также попутно выполнять сравнение *каталогов*. Начнем с реализации сравнения имен файлов для одного каталога, представленной в примере 6.11, так как это вложенная и более простая операция.

Пример 6.11. PP4E\System\Filetools\dirdiff.py

```
"""
################################################################################
Порядок использования: python dirdiff.py dir1-path dir2-path
Сравнивает два каталога, пытаясь отыскать файлы, присутствующие в одном
и отсутствующие в другом.
Эта версия использует функцию os.listdir и выполняет поиск различий между двумя
списками. Обратите внимание, что сценарий проверяет только имена файлов, но не
их содержимое, – версию, которая сравнивает результаты вызова методов .read(), вы
найдете в сценарии diffall.py.
################################################################################
"""

import os, sys

def reportdiffs(unique1, unique2, dir1, dir2):
    """
    Генерирует отчет о различиях для одного каталога: часть вывода функции
    comparedirs
    """
    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 difference(seq1, seq2):
    """
    Возвращает элементы, присутствующие только в seq1;
    Операция set(seq1) - set(seq2) даст аналогичный результат, но множества
    являются неупорядоченными коллекциями, поэтому порядок следования
    элементов в каталоге будет утерян
    """
    return [item for item in seq1 if item not in seq2]

def comparedirs(dir1, dir2, files1=None, files2=None):
    """
    Сравнивает содержимое каталогов, но не сравнивает содержимое файлов;
    функции listdir может потребоваться передавать аргумент типа bytes, если
    могут встречаться имена файлов, недекодируемые на других платформах
    """
    print('Comparing', dir1, 'to', dir2)
    files1 = os.listdir(dir1) if files1 is None else files1
    files2 = os.listdir(dir2) if files2 is None else files2
    unique1 = difference(files1, files2)
    unique2 = difference(files2, files1)
    reportdiffs(unique1, unique2, dir1, dir2)
    return not (unique1 or unique2)            # true, если нет различий

def getargs():
    "Аргументы при работе в режиме командной строки"
    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 о файлах, имеющихся в одном каталоге, но отсутствующих в другом). Функция `comparedirs` возвращает значение `True`, если расхождения не были обнаружены, что полезно для обнаружения различий при вызове из других программ.

Запустим этот сценарий с несколькими каталогами – он сообщит о найденных различиях, которые представляют уникальные имена в любом из переданных каталогов. Обратите внимание, что при этом сравниваются только структуры путем проверки имен в списках, но не содержимое файлов (эта возможность будет добавлена ниже):

```
C:\...\PP4E\System\Filetools> dirdiff.py C:\temp\PP3E\Examples copytemp
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical

C:\...\PP4E\System\Filetools> dirdiff.py C:\temp\PP3E\Examples\PP3E\System ..
Comparing C:\temp\PP3E\Examples\PP3E\System to ..
Files unique to C:\temp\PP3E\Examples\PP3E\System
... App
... Exits
... Media
... moreplus.py
Files unique to ..
... more.pyc
... spam.txt
... Tester
... __init__.pyc
```

В основе сценария лежит функция `difference`: она реализует простую операцию сравнения списков. Применительно к каталогам, *уникальные* элементы представляют различия между деревьями, а *общие* элементы представляют имена файлов или подкаталогов, которые заслуживают дальнейшего сравнения или обхода. В Python 2.4 и более поздних версиях можно было бы использовать встроенные объекты типа `set`, если порядок следования имен в результатах не имеет значения – множества не являются последовательностями, поэтому они не сохраняют оригинальный порядок следования элементов в списках, полученных с помощью функции `os.listdir`. По этой причине (и чтобы не вынуждать пользователей модернизировать сценарий) вместо множеств мы будем использовать функцию, опирающуюся на использование выражения-генератора.

Поиск различий между деревьями

Мы только что реализовали инструмент, отбирающий уникальные имена файлов и каталогов. Теперь нам осталось реализовать инструмент обхода дерева, который будет применять функции из модуля `dirdiff` на каждом уровне, чтобы отобрать уникальные файлы и каталоги; явно сравнит содержимое общих файлов и обойдет общие каталоги. Эти операции осуществляет сценарий из примера 6.12.

Пример 6.12. PP4E\System\Filetools\diffall.py

```
"""
################################################################################
Порядок использования: "python diffall.py dir1 dir2".
Выполняет рекурсивное сравнение каталогов: сообщает об уникальных файлах,
существующих только в одном из двух каталогов, dir1 или dir2; сообщает о файлах
с одинаковыми именами и с разным содержимым, присутствующих в каталогах dir1
и dir2; сообщает об разнотипных элементах с одинаковыми именами, присутствующих
```

```
в каталогах dir1 и dir2; то же самое выполняется для всех подкаталогов
с одинаковыми именами, находящихся внутри деревьев каталогов dir1
и dir2. Сводная информация об обнаруженных отличиях помещается в конец вывода,
однако в процессе поиска в вывод добавляется дополнительная информация об
отличающихся и уникальных файлах с метками "DIFF" и "unique". Новое: (в 3
издании) для больших файлов введено ограничение на размер читаемых блоков в 1
Мбайт, (3 издание) обнаруживаются одинаковые имена файлов/каталогов, (4 издание)
исключены лишние вызовы os.listdir() в dirdiff.comparedirs() за счет передачи
результатов.
##############################################################################
"""

import os, dirdiff
blocksize = 1024 * 1024            # не более 1 Мбайта на одну операцию чтения

def intersect(seq1, seq2):
    """
    Возвращает все элементы, присутствующие одновременно в seq1 и seq2;
    выражение set(seq1) & set(seq2) возвращает тот же результат, но множества
    являются неупорядоченными коллекциями, поэтому при их использовании может
    быть утерян порядок следования элементов, если он имеет значение для
    некоторых платформ
    """
    return [item for item in seq1 if item in seq2]

def comparetrees(dir1, dir2, diffs, verbose=False):
    """
    Сравнивает все подкаталоги и файлы в двух деревьях каталогов;
    для предотвращения кодирования/декодирования содержимого и преобразования
    символов конца строки использует двоичный режим доступа к файлам,
    так как деревья могут содержать произвольные двоичные и текстовые файлы;
    функции listdir может потребоваться передавать аргумент типа bytes, если
    могут встречаться имена файлов, недекодируемые на других платформах
    """
    # сравнить списки с именами файлов
    print('-' * 20)
    names1 = os.listdir(dir1)
    names2 = os.listdir(dir2)
    if not dirdiff.comparedirs(dir1, dir2, names1, names2):
        diffs.append('unique files at %s - %s' % (dir1, dir2))

    print('Comparing contents')
    common = intersect(names1, names2)
    missed = common[:]

    # сравнить содержимое файлов с одинаковыми именами
    for name in common:
        path1 = os.path.join(dir1, name)
        path2 = os.path.join(dir2, name)
        if os.path.isfile(path1) and os.path.isfile(path2):
            missed.remove(name)
```

```
                file1 = open(path1, 'rb')
                file2 = open(path2, 'rb')
                while True:
                    bytes1 = file1.read(blocksize)
                    bytes2 = file2.read(blocksize)
                    if (not bytes1) and (not bytes2):
                        if verbose: print(name, 'matches')
                        break
                    if bytes1 != bytes2:
                        diffs.append('files differ at %s - %s' % (path1, path2))
                        print(name, 'DIFFERS')
                        break

        # рекурсивно сравнить каталоги с одинаковыми именами
        for name in common:
            path1 = os.path.join(dir1, name)
            path2 = os.path.join(dir2, name)
            if os.path.isdir(path1) and os.path.isdir(path2):
                missed.remove(name)
                comparetrees(path1, path2, diffs, verbose)

        # одинаковые имена, но оба не являются одновременно файлами или каталогами?
        for name in missed:
            diffs.append('files missed at %s - %s: %s' % (dir1, dir2, name))
            print(name, 'DIFFERS')

    if __name__ == '__main__':
        dir1, dir2 = dirdiff.getargs()
        diffs = []
        comparetrees(dir1, dir2, diffs, True)  # список diffs изменяется в
        print('=' * 40)                        # процессе обхода, вывести diffs
        if not diffs:
            print('No diffs found.')
        else:
            print('Diffs found:', len(diffs))
            for diff in diffs: print('-', diff)
```

В каждом каталоге этого дерева данный сценарий просто использует модуль `dirdiff`, чтобы обнаружить уникальные имена, а затем сравнивает общие имена, присутствующие одновременно в обоих списках содержимого каталогов. Рекурсивный спуск в подкаталоги выполняется только после сравнения всех файлов на каждом уровне, чтобы вывод сценария было удобнее воспринимать на глаз (трассировка обхода подкаталогов выводится ниже результатов сравнения файлов – они не смешиваются).

Обратите внимание на список `misses`, добавленный в сценарий в третьем издании книги. Очень маловероятно, но не невозможно, чтобы одно и то же имя в одном каталоге соответствовало файлу, а в другом – подкаталогу. Кроме того, обратите внимание на переменную `blocksize`. Как и в сценарии копирования деревьев каталогов, который мы видели выше,

вместо того чтобы слепо пытаться читать файлы в память целиком, мы установили ограничение в 1 Мбайт для каждой операции чтения – на тот случай, если какие-нибудь файлы окажутся слишком большими, чтобы их можно было загрузить в память. Предыдущая версия этого сценария, без этого ограничения, которая читала файлы целиком, как показано ниже, на некоторых компьютерах возбуждала исключение `MemoryError`:

```
bytes1 = open(path1, 'rb').read()
bytes2 = open(path2, 'rb').read()
if bytes1 == bytes2: ...
```

Этот код проще, но менее практичен в ситуациях, когда могут встречаться очень большие файлы, не умещающиеся в память целиком (например, файлы образов CD и DVD). В новой версии файл читается в цикле порциями не более 1 Мбайта, пока не будет возвращена пустая строка, свидетельствующая об окончании файла. Файлы считаются одинаковыми, если совпадают все прочитанные из них блоки и конец файла достигнут одновременно.

Помимо всего прочего, мы обрабатываем содержимое файлов в двоичном режиме, чтобы подавить операцию декодирования их содержимого и предотвратить преобразование символов конца строки, потому что деревья каталогов могут содержать произвольные двоичные и текстовые файлы. Держим также наготове обычное замечание о необходимости передачи аргумента типа `bytes` функции `os.listdir` на платформах, где имена файлов могут оказаться недекодируемыми (например, с помощью `dir1.encode()`). На некоторых платформах может также потребоваться определять и пропускать некоторые файлы специальных типов, чтобы обеспечить полную универсальность, но в моих каталогах такие файлы отсутствовали, поэтому я не включил эту проверку в сценарий.

В четвертом издании было внесено одно незначительное изменение: для каждого подкаталога результаты `os.listdir` теперь собираются и передаются только один раз, чтобы избежать лишних вызовов функций из модуля `dirdiff`, – не самая большая победа, но на моем медлительном нетбуке, который я использовал во время работы над этим изданием, каждый лишний цикл на счету.

Запускаем сценарий

Так как мы уже изучали инструменты обхода деревьев, задействованные в этом сценарии, перейдем прямо к некоторым примерам его использования. При обработке идентичных деревьев во время обхода выводятся сообщения о состоянии, а в конце появляется сообщение: «No diffs found» (Расхождений не обнаружено):

```
C:\...\PP4E\System\Filetools> diffall.py C:\temp\PP3E\Examples
                                         copytemp > diffs.txt
C:\...\PP4E\System\Filetools> type diffs.txt | more
--------------------
```

```
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical
Comparing contents
README-root.txt matches
--------------------
Comparing C:\temp\PP3E\Examples\PP3E to copytemp\PP3E
Directory lists are identical
Comparing contents
echoEnvironment.pyw matches
LaunchBrowser.pyw matches
Launcher.py matches
Launcher.pyc matches
...более 2000 строк опущено...
--------------------
Comparing C:\temp\PP3E\Examples\PP3E\TempParts to copytemp\PP3E\TempParts
Directory lists are identical
Comparing contents
109_0237.JPG matches
lawnlake1-jan-03.jpg matches
part-001.txt matches
part-002.html matches
========================================
No diffs found.
```

При использовании этого сценария я обычно устанавливаю флаг `verbose` в значение `True` и перенаправляю вывод в файл (для больших деревьев выводится слишком много информации, которую трудно воспринимать в процессе выполнения сценария). Чтобы ограничить количество сообщений, устанавливайте флаг `verbose` в значение `False`. Чтобы посмотреть, как выглядит отчет о расхождениях, нужно их создать. Для простоты я вручную изменил несколько файлов в одном из деревьев, но вы можете воспользоваться сценарием глобального поиска и замены, представленным выше в этой главе. Заодно удалим несколько файлов, чтобы в процессе поиска можно было обнаружить уникальные элементы. Последние две команды удаления из приведенных ниже воздействуют на один и тот же каталог в разных деревьях:

```
C:\...\PP4E\System\Filetools> notepad copytemp\PP3E\README-PP3E.txt
C:\...\PP4E\System\Filetools> notepad copytemp\PP3E\System\Filetools\commands.py
C:\...\PP4E\System\Filetools> notepad C:\temp\PP3E\Examples\PP3E\__init__.py

C:\...\PP4E\System\Filetools> del copytemp\PP3E\System\Filetools\cpall_visitor.py
C:\...\PP4E\System\Filetools> del copytemp\PP3E\Launcher.py
C:\...\PP4E\System\Filetools> del C:\temp\PP3E\Examples\PP3E\PyGadgets.py
```

Теперь перезапустим сценарий сравнения, чтобы обнаружить различия, и перенаправим вывод в файл, чтобы облегчить просмотр результатов. 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, – конечно, этот интерфейс можно было бы сделать более дружественным, но мне вполне хватает и этого:

```
C:\...\PP4E\System\Filetools> diffall.py C:\temp\PP3E\Examples
                                         copytemp > diff2.txt
C:\...\PP4E\System\Filetools> notepad diff2.txt
--------------------
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical
Comparing contents
README-root.txt matches
--------------------
Comparing C:\temp\PP3E\Examples\PP3E to copytemp\PP3E
Files unique to C:\temp\PP3E\Examples\PP3E
... Launcher.py
Files unique to copytemp\PP3E
... PyGadgets.py
Comparing contents
echoEnvironment.pyw matches
LaunchBrowser.pyw matches
Launcher.pyc matches
...множество строк опущено...
PyGadgets_bar.pyw matches
README-PP3E.txt DIFFERS
todos.py matches
tounix.py matches
__init__.py DIFFERS
__init__.pyc matches
--------------------
Comparing C:\temp\PP3E\Examples\PP3E\System\Filetools to copytemp\PP3E\System\
Fil...
Files unique to C:\temp\PP3E\Examples\PP3E\System\Filetools
... cpall_visitor.py
Comparing contents
commands.py DIFFERS
cpall.py matches
...множество строк опущено...
--------------------
Comparing C:\temp\PP3E\Examples\PP3E\TempParts to copytemp\PP3E\TempParts
Directory lists are identical
Comparing contents
109_0237.JPG matches
lawnlake1-jan-03.jpg matches
part-001.txt matches
part-002.html matches
========================================
Diffs found: 5
- unique files at C:\temp\PP3E\Examples\PP3E - copytemp\PP3E
- files differ at C:\temp\PP3E\Examples\PP3E\README-PP3E.txt -
```

```
        copytemp\PP3E\README-PP3E.txt
- files differ at C:\temp\PP3E\Examples\PP3E\__init__.py -
        copytemp\PP3E\__init__.py
- unique files at C:\temp\PP3E\Examples\PP3E\System\Filetools -
        copytemp\PP3E\System\Filetools
- files differ at C:\temp\PP3E\Examples\PP3E\System\Filetools\commands.py -
        copytemp\PP3E\System\Filetools\commands.py
```

Я добавил разрывы строк и отступы кое-где, чтобы уместить листинг по ширине страницы, но отчет легко понять. В дереве, насчитывающем 1430 файлов и 185 каталогов, было найдено пять различий – три файла были изменены мною вручную, а два каталога мы рассогласовали тремя командами удаления.

Проверка резервных копий

Каким же образом этот сценарий способен унять паранойю при создании резервных копий на CD? Для дублирующей проверки работы моего пишущего привода CD я выполняю команду, как показано ниже. С помощью такой команды я могу также найти изменения, произведенные после предыдущего резервного копирования. И снова, поскольку на моем компьютере привод CD представляется как «G:», я указываю путь с таким корнем. В Linux используйте корень вида */dev/cdrom* или */mnt/cdrom*:

```
C:\...\PP4E\System\Filetools> python diffall.py Examples
                                        g:\PP3E\Examples > diff0226
C:\...\PP4E\System\Filetools> more diff0226
...вывод опущен...
```

Компакт-диск крутится, сценарий сравнивает, и в конце отчета появляется информация о различиях. Пример полного отчета о различиях находится в файле *diff*.txt* в пакете с примерами для этой книги. А чтобы быть *действительно* уверенным, я выполняю следующую команду глобального сравнения – чтобы проверить все дерево с резервной копией книги на флешке (которая, с точки зрения файловой системы, ничем не отличается от CD):

```
C:\...\PP4E\System\Filetools> diffall.py F:\writing-backups\feb-26-10\dev
                              C:\Users\mark\Stuff\Books\4E\PP4E\dev > diff3.txt
C:\...\PP4E\System\Filetools> more diff3.txt
--------------------
Comparing F:\writing-backups\feb-26-10\dev to C:\Users\mark\Stuff\Books\4E\PP4E\
dev
Directory lists are identical
Comparing contents
ch00.doc DIFFERS
ch01.doc matches
ch02.doc DIFFERS
ch03.doc matches
ch04.doc DIFFERS
```

```
ch05.doc matches
ch06.doc DIFFERS
...множество строк опущено...
--------------------
Comparing F:\writing-backups\feb-26-10\dev\Examples\PP4E\System\Filetools to
C:\...
Files unique to C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\
Filetools
... copytemp
... cpall.py
... diff2.txt
... diff3.txt
... diffall.py
... diffs.txt
... dirdiff.py
... dirdiff.pyc
Comparing contents
bigext-tree.py matches
bigpy-dir.py matches
...множество строк опущено...
========================================
Diffs found: 7
- files differ at F:\writing-backups\feb-26-10\dev\ch00.doc -
        C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch00.doc
- files differ at F:\writing-backups\feb-26-10\dev\ch02.doc -
        C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch02.doc
- files differ at F:\writing-backups\feb-26-10\dev\ch04.doc -
        C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch04.doc
- files differ at F:\writing-backups\feb-26-10\dev\ch06.doc -
        C:\Users\mark\Stuff\Books\4E\PP4E\dev\ch06.doc
- files differ at F:\writing-backups\feb-26-10\dev\TOC.txt -
        C:\Users\mark\Stuff\Books\4E\PP4E\dev\TOC.txt
- unique files at F:\writing-backups\feb-26-10\dev\Examples\PP4E\System\Filetools -
        C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\System\Filetools
- files differ at F:\writing-backups\feb-26-10\dev\Examples\PP4E\Tools\visitor.py -
        C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Tools\visitor.py
```

Данный пример показывает, что после создания предыдущей резервной копии я добавил несколько примеров и изменил несколько файлов с текстом глав. Если выполнить эту команду сразу после создания резервной копии, то радаром `diffall` не будет обнаружено никаких изменений, за исключением файлов, которые вообще не могут быть скопированы. Такое глобальное сравнение может занять несколько минут. Эта операция выполняет побайтовое сравнение всех файлов глав и снимков с экрана, дерева примеров и других, но это точное и полное сравнение. Дерево каталогов этой книги содержит большое количество файлов, и какая-либо менее автоматизированная процедура проверки без помощи Python была бы совершенно немыслима.

После того как этот сценарий был написан, я начал использовать его для проверки резервных копий моих ноутбуков на внешнем жестком диске, создаваемых автоматически. Для этого я запускаю сценарий `cpall`, написанный нами в предыдущем разделе этой главы, а затем, чтобы проверить результаты и получить список файлов, вызвавших проблемы при копировании, – сценарий сравнения, разработанный здесь. Когда я выполнял эту процедуру в последний раз, было скопировано и проверено 225 000 файлов и 15 000 каталогов, занимающих 20 Гбайт дискового пространства, – это явно не та задача, которую можно выполнить вручную!

Ниже приводятся магические заклинания, которые я вводил на моем ноутбуке с системой Windows. Здесь `f:\` – это раздел на внешнем жестком диске, и вас не должно удивлять, что каждая из этих команд выполняется около получаса или даже больше на распространенном аппаратном обеспечении. Копирование, инициированное операцией перетаскивания мышью, выполняется ничуть не быстрее (если вообще выполняется!):

```
C:\...\System\Filetools> cpall.py c:\ f:\ > f:\copy-log.txt
C:\...\System\Filetools> diffall.py f:\ c:\ > f:\diff-log.txt
```

Отчет о различиях и другие идеи

Наконец, стоит заметить, что этот сценарий *обнаруживает* расхождения только в деревьях, не сообщая никаких подробностей о различиях в отдельных файлах. На самом деле он просто загружает и сравнивает двоичное содержимое соответствующих файлов в виде строк, давая простой результат «да/нет».

Когда мне нужны дополнительные сведения о фактических различиях в двух несовпавших 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 стоявших передо мною задач просто нахождение различий в дереве из 1400 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). Вывод этих системных вызовов можно было бы поместить в отчет в первоначальном виде или оставить только наиболее важные его части.

Мы могли бы также открывать текстовые файлы в текстовом режиме, чтобы игнорировать различия, вызванные разными комбинациями символов завершения строк при передаче файлов между платформами,

но не всегда ясно – действительно ли такие отличия должны игнорироваться (что если пользователь пожелает узнать, не изменились ли символы конца строки?). Например, после загрузки файла с веб-сайта с помощью сценария FTP, с которым мы встретимся в главе 13, сценарий `diffall` обнаруживает несоответствие между локальной копией файла и оригиналом на удаленном сервере. Продолжая исследования, я просто выполнил несколько инструкций в интерактивном сеансе Python:

```
>>> a = open('lp2e-updates.html', 'rb').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'rb').read()
>>> a == b
False
```

Эта проверка показывает, что двоичное содержимое локальной версии файла отличается от содержимого удаленной версии. Чтобы выяснить, обусловлено ли это различием способов завершения строк в Unix и DOS, я попробовал выполнить то же самое, но в текстовом режиме, чтобы перед сравнением символы окончания строк были приведены к стандартному символу `\n`:

```
>>> a = open('lp2e-updates.html', 'r').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'r').read()
>>> a == b
True
```

Теперь, чтобы отыскать различия, я выполнил следующие инструкции, которые проверяют содержимое символ за символом, пока не наткнутся на первое несоответствие (применение двоичного режима сохраняет различия):

```
>>> a = open('lp2e-updates.html', 'rb').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'rb').read()
>>> for (i, (ac, bc)) in enumerate(zip(a, b)):
...     if ac != bc:
...         print(i, repr(ac), repr(bc))
...         break
...
37966 '\r' '\n'
```

Этот результат означает, что в загруженном файле в байте со смещением 37 966 находится символ `\r`, а в локальной копии – символ `\n`. Эта строка в одном файле оканчивается комбинацией символов завершения строки в DOS, а в другом – символом завершения строки в Unix. Чтобы увидеть больше, можно вывести текст, окружающий несовпадение:

```
>>> for (i, (ac, bc)) in enumerate(zip(a, b)):
...     if ac != bc:
...         print(i, repr(ac), repr(bc))
...         print(repr(a[i-20:i+20]))
...         print(repr(b[i-20:i+20]))
...         break
...
```

```
37966 '\r' '\n'
're>\r\ndef min(*args):\r\n     tmp = list(arg'
're>\r\ndef min(*args):\n     tmp = list(args'
```

По всей видимости, я вставил символ завершения строки Unix в одном месте в локальной копии, там, где в загруженной версии находится комбинация символов завершения строки в DOS, – результат использования текстового режима в сценарии загрузки (который преобразует символы `\n` в комбинации `\r\n`) и многих лет использования ноутбуков и PDA, работающих под управлением Linux и Windows (вероятно, я внес это изменение, когда после редактирования этого файла в Linux я скопировал его в Windows в двоичном режиме). Такой программный код, как показано выше, можно было бы добавить в сценарий `diffall`, чтобы обеспечить более интеллектуальное сравнение текстовых файлов и вывод более подробной информации об отличиях в них.

Поскольку Python отлично подходит для обработки строк и файлов, можно пойти еще дальше и реализовать на языке Python сценарий, эквивалентный командам `fc` и `diff`. Фактически большая часть работы в этом направлении уже выполнена – эту задачу можно было бы существенно упростить, задействовав модуль `difflib` из стандартной библиотеки. Более подробные сведения о нем и примеры использования вы найдете в руководстве по библиотеке Python.

Можно было бы поступить еще умнее и не выполнять загрузку и сравнение файлов, отличающихся размерами; выполнять чтение файлов более мелкими порциями, чтобы уменьшить потребление памяти. Для большинства деревьев каталогов такие оптимизации излишни – чтение многомегабайтных файлов в строки в Python осуществляется очень быстро, а память постоянно освобождается сборщиком мусора в процессе работы.

Поскольку эти улучшения выходят за рамки задач данного сценария и размеров этой главы, его предоставляется выполнить любознательному читателю. (В этой книге официально отсутствуют упражнения для самостоятельного решения, но данное предложение выглядит именно так, не правда ли?) А теперь перейдем к исследованию способов реализации еще одной операции, часто применяемой к деревьям каталогов: поиск.

Поиск в деревьях каталогов

Инженеры любят что-нибудь изменять. 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 начинало казаться, что я набрел на более подходящую структуру. На достаточно раннем этапе все было приемлемо, но по мере того как дерево становилось все более запутанным, сопровождение его превратилось в кошмар. Пути к каталогам

программ и имена модулей были разбросаны по всему программному коду – в операциях импорта пакетов, вызовах программ, комментариях, файлах конфигурации и так далее.

Можно, конечно, исправлять все эти ссылки, вручную редактируя файлы в каталоге и отыскивая в них ту информацию, которая изменилась. Однако это настолько объемная работа, что для дерева примеров данной книги она практически неосуществима. Дерево примеров к предыдущему изданию содержит 186 каталогов и 1429 файлов! Очевидно, что мне был необходим инструмент, автоматизирующий обновление после производимых изменений. Существуют различные способы решения этой задачи – от команд оболочки до операций поиска, функций обхода деревьев и универсальных фреймворков. В этом и в следующем разделах мы исследуем каждый способ в отдельности, как это сделал я в поисках наиболее оптимального решения этой проблемы.

grep, glob и find

Если вы работаете в Unix-подобной системе, то наверняка знаете о существовании стандартного способа поиска строк в файлах в таких системах. Программа командной строки `grep` и родственные ей позволяют получить перечень всех строк в одном или нескольких файлах, содержащих строку или шаблон строки.[1] Учитывая, что командные оболочки Unix автоматически расширяют (то есть «глобализуют») шаблоны имен файлов, такая команда, как приведена ниже, будет искать строку, указанную в командной строке, в файлах Python, расположенных в одном каталоге (в этом примере используется команда `grep`, входящая в состав облочки Cygwin для Windows, о которой я рассказывал в предыдущей главе):

```
C:\...\PP4E\System\Filetools> c:\cygwin\bin\grep.exe walk *.py
bigext-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):
bigpy-path.py:     for (thisDir, subsHere, filesHere) in os.walk(srcdir):
bigpy-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):
```

Как мы уже знаем, те же действия можно запрограммировать в сценарии на языке Python, организовав запуск такой команды с помощью `os.system` или `os.popen`. А в случае реализации операции поиска вручную, мы могли бы добиться похожих результатов с помощью модуля `glob`, с которым познакомились в главе 4, – он, подобно командной оболочке, расширяет шаблоны имен файлов в списки строк соответствующих имен файлов:

```
C:\...\PP4E\System\Filetools> python
>>> import os
>>> for line in os.popen(r'c:\cygwin\bin\grep.exe walk *.py'):
...     print(line, end='')
```

[1] Среди разработчиков, проведших в гетто Unix достаточное время, операция поиска в файлах часто носит разговорное название «grepping».

```
...
bigext-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):
bigpy-path.py:     for (thisDir, subsHere, filesHere) in os.walk(srcdir):
bigpy-tree.py:for (thisDir, subsHere, filesHere) in os.walk(dirname):

>>> from glob import glob
>>>     for filename in glob('*.py'):
...         if 'walk' in open(filename).read():
...             print(filename)
...
bigext-tree.py
bigpy-path.py
bigpy-tree.py
```

К сожалению, область действия этих инструментов обычно ограничивается одним каталогом. Модуль `glob` способен выполнить обход нескольких каталогов при правильно сформированной строке шаблона, но он не является универсальным средством обхода деревьев каталогов, который требуется мне для обслуживания большого дерева каталогов с примерами. В Unix-подобных системах команда `find` оболочки предоставляет расширенные возможности для обхода всего дерева каталогов. Например, следующая команда Unix точно определила бы файлы и строки в текущем каталоге и ниже, где встречается строка `popen`:

```
find . -name "*.py" -print -exec fgrep popen {} \;
```

Если у вас имеется Unix-подобная команда `find` на всех компьютерах, которыми вы пользуетесь, можете считать, что у вас есть инструмент для обработки каталогов.

Создание собственного модуля find

Но если команда `find` доступна не на всех ваших компьютерах, не волнуйтесь – ее легко можно реализовать на языке Python. Ранее в стандартной библиотеке Python имелся модуль `find`, который я часто использовал. И хотя этот модуль был удален из библиотеки где-то между вторым и третьим изданиями этой книги, в стандартной библиотеке появилась функция `os.walk`, которая способна упростить создание собственного модуля `find`. Вместо того чтобы оплакивать исчезновение модуля, я решил потратить 10 минут и написать свой эквивалент.

В примере 6.13 представлена утилита `find`, реализованная на языке Python, которая выбирает все имена файлов в каталоге, соответствующие шаблону. В отличие от `glob.glob`, функция `find.find` автоматически выполняет поиск во всем дереве каталогов. А в отличие от структуры обхода `os.walk`, результаты `find.find` можно трактовать, как простую линейную группу строк.

Пример 6.13. PP4E\Tools\find.py

```
#!/usr/bin/python
"""
```

```
###############################################################################
Возвращает все имена файлов, соответствующие шаблону в дереве каталогов;

собственная версия модуля find, ныне исключенного из стандартной библиотеки:
импортируется как "PP4E.Tools.find"; похож на оригинал, но использует цикл
os.walk, не поддерживает возможность обрезания ветвей подкаталогов и может
запускаться как самостоятельный сценарий;

find() - функция-генератор, использующая функцию-генератор os.walk(),
возвращающая только имена файлов, соответствующие шаблону: чтобы получить весь
список результатов сразу, используйте функцию findlist();
###############################################################################
"""

import fnmatch, os

def find(pattern, startdir=os.curdir):
    for (thisDir, subsHere, filesHere) in os.walk(startdir):
        for name in subsHere + filesHere:
            if fnmatch.fnmatch(name, pattern):
                fullpath = os.path.join(thisDir, name)
                yield fullpath

def findlist(pattern, startdir=os.curdir, dosort=False):
    matches = list(find(pattern, startdir))
    if dosort: matches.sort()
    return matches

if __name__ == '__main__':
    import sys
    namepattern, startdir = sys.argv[1], sys.argv[2]
    for name in find(namepattern, startdir): print(name)
```

Вроде бы немного делает этот программный код – по сути, он лишь несколько расширяет возможности функции `os.walk`, – но его функция `find` позволяет получить те же результаты, что давал ранее существовавший в стандартной библиотеке модуль `find` и одноименная утилита в Unix. Кроме того, этот модуль является более переносимым решением, и пользоваться им намного проще, чем повторять его программный код всякий раз, когда потребуется выполнить поиск. Поскольку этот файл можно использовать и как сценарий, и как библиотечный модуль, его можно применять как инструмент командной строки и вызывать из других программ.

Например, чтобы обработать все файлы с программным кодом на языке Python, находящиеся в дереве каталогов, с корнем на один уровень выше текущего рабочего каталога, достаточно просто запустить приведенную ниже команду в окне консоли. Запустите эту команду у себя, чтобы увидеть ее в действии, – в данном примере стандартный вывод сценария передается по конвейеру команде `more`, обеспечивающей возможность постраничного просмотра результатов, но его точно так же

можно передать любой другой программе обработки, которая читает входные данные из стандартного потока ввода:

```
C:\...\PP4E\Tools> python find.py *.py .. | more
..\LaunchBrowser.py
..\Launcher.py
..\__init__.py
..\Preview\attachgui.py
..\Preview\customizegui.py
```

...множество строк опущено...

Чтобы получить еще больший контроль, выполните следующий программный код на языке Python в сценарии или в интерактивной оболочке. При таком подходе к найденным файлам можно применять любые операции, доступные в языке Python:

```
C:\...\PP4E\System\Filetools> python
>>> from PP4E.Tools import find              # или просто import find, если
>>> for filename in find.find('*.py', '..'): # модуль находится в cwd
...     if 'walk' in open(filename).read():
...         print(filename)
...
..\Launcher.py
..\System\Filetools\bigext-tree.py
..\System\Filetools\bigpy-path.py
..\System\Filetools\bigpy-tree.py
..\Tools\cleanpyc.py
..\Tools\find.py
..\Tools\visitor.py
```

Обратите внимание, что в данном случае отпала необходимость во вложенных циклах, необходимых при использовании функции `os.walk`, когда требуется получить список имен файлов, соответствующих шаблону, – во многих случаях такой подход концептуально выглядит проще. Заметьте также, что функция поиска является функцией-генератором, благодаря чему сценарию не приходится ждать, пока не будут выбраны все соответствующие имена файлов, – функция `os.walk` поставляет результаты для каждого каталога в отдельности, а функция `find.find` поставляет имена файлов, выбирая их из этих результатов.

Ниже приводится более сложный пример использования модуля `find`: следующая команда выводит все имена файлов с программным кодом на языке Python, находящиеся в дереве каталогов с корнем в *C:\temp\PP3E*, начинающиеся с символа *q* или *t*. Обратите внимание, что `find` возвращает полные пути к файлам, начиная от указанного каталога:

```
C:\...\PP4E\Tools> find.py [qx]*.py C:\temp\PP3E
C:\temp\PP3E\Examples\PP3E\Database\SQLscripts\querydb.py
C:\temp\PP3E\Examples\PP3E\Gui\Tools\queuetest-gui-class.py
C:\temp\PP3E\Examples\PP3E\Gui\Tools\queuetest-gui.py
C:\temp\PP3E\Examples\PP3E\Gui\Tour\quitter.py
C:\temp\PP3E\Examples\PP3E\Internet\Other\Grail\Question.py
```

```
C:\temp\PP3E\Examples\PP3E\Internet\Other\XML\xmlrpc.py
C:\temp\PP3E\Examples\PP3E\System\Threads\queuetest.py
```

А ниже приводится программный код Python, который выполняет тот же поиск, но при этом выводит только имена файлов без путей к ним и размеры файлов:

```
C:\...\PP4E\Tools> python
>>> import os
>>> from find import find
>>> for name in find('[qx]*.py', r'C:\temp\PP3E'):
...     print(os.path.basename(name), os.path.getsize(name))
...
querydb.py 635
queuetest-gui-class.py 1152
queuetest-gui.py 963
quitter.py 801
Question.py 817
xmlrpc.py 705
queuetest.py 1273
```

Модуль fnmatch

Чтобы добиться такой экономии программного кода, модуль `find` вызывает функцию `os.walk` для обхода дерева каталогов и просто возвращает соответствующие имена файлов в процессе обхода. Однако в нем содержится еще одна новинка – модуль `fnmatch`, входящий в состав стандартной библиотеки Python, который выполняет сопоставление имен файлов с шаблоном. Этот модуль поддерживает общие операторы в строках шаблонов: `*` соответствует любому количеству символов, `?` соответствует одному любому символу, а `[...]` и `[!...]` соответствуют любым символам, перечисленным и отсутствующим в квадратных скобках, соответственно; другие символы соответствуют самим себе. В отличие от модуля `re`, модуль `fnmatch` поддерживает только самые общие операторы шаблонов командной оболочки Unix и не поддерживает полноценные регулярные выражения. Значение этого отличия мы увидим в главе 19.

Интересно отметить, что функция `glob.glob` тоже использует модуль `fnmatch` для сопоставления имен: она объединяет `os.listdir` и `fnmatch` для сопоставления имен файлов в каталоге практически так же, как наша функция `find.find` объединяет `os.walk` и `fnmatch` для поиска совпадений в деревьях (хотя функция `os.walk`, в свою очередь, использует функцию `os.listdir`). Одно из следствий всего этого состоит в том, что имеется возможность передавать функции `find.find` имя начального каталога и шаблон в виде строк байтов, если необходимо подавить декодирование имен файлов, содержащих символы Юникода, как это возможно при использовании функций `os.walk` и `glob.glob`, – в результате вы будете получать имена файлов в виде строк байтов. Подробнее о символах Юникода в именах файлов рассказывается в главе 4.

Для сравнения, вызов find.find со строкой шаблона «*» является примерным эквивалентом команды оболочки, выводящей содержимое дерева каталогов, такой как dir /B /S в DOS и Windows. Поскольку шаблону «*» соответствуют все файлы, такой вызов вернет все имена файлов, присутствующих в дереве, за один проход. Подобные команды мы легко можем выполнять в сценариях на языке Python с помощью функции os.popen, поэтому следующий фрагмент выполняет ту же самую работу, но он изначально является непереносимым и приводит к запуску параллельной программы:

```
>>> import os
>>> for line in os.popen('dir /B /S'): print(line, end='')

>>> from PP4E.Tools.find import find
>>> for name in find(pattern='*', startdir='.'): print(name)
```

Данная утилита еще будет демонстрироваться далее в этой главе и в книге, включая самые, пожалуй, убедительные демонстрации в следующем разделе и в диалоге Grep, в главе 11 – в реализации текстового редактора PyEdit с графическим интерфейсом, где она будет играть центральную роль в многопоточном внешнем инструменте поиска. Модуль find был исключен из стандартной библиотеки, но это не повод забывать о нем.

Если модулю fnmatch имя файла передается в виде строки байтов, то шаблон также *должен* иметь тип bytes (либо оба аргумента должны иметь тип str), потому что используемый им модуль re, реализующий сопоставление с регулярными выражениями, не позволяет смешивать типы испытуемой строки и шаблона. Это требование по наследству переходит и нашей функции find.find, принимающей имя каталога и шаблон. Подробнее о модуле re рассказывается в главе 19.

Любопытно отметить, что модуль fnmatch в Python 3.1 также преобразует строку шаблона типа bytes в строку str Юникода и обратно в ходе внутренней обработки текста, используя при этом кодировку Latin-1. Этого достаточно для большинства применений, но это может вступать в противоречие с некоторыми кодировками, которые неточно отображаются в кодировку Latin-1. В таких ситуациях параметр sys.getfilesystemencoding мог бы точнее соответствовать используемой кодировке, так как он отражает ограничения, накладываемые файловой системой (как мы узнали в главе 4, параметр sys.getdefaultencoding отражает кодировку содержимого файлов, а не их имен).

Когда функции os.walk передаются аргументы типа str, она предполагает, что имена файлов следуют соглашениям для данной платформы, и не игнорирует ошибки декодирования, возбуждаемые функцией os.listdir. В утилите «grep» в примере PyEdit в главе 11 эта картина еще больше омрачается

тем фактом, что строку str шаблона, полученную из графического интерфейса, необходимо кодировать в строку bytes, используя кодировку, возможно, неподходящую для некоторых файлов. За дополнительными подробностями обращайтесь к описанию функций fnmatch.py и os.py в руководстве по стандартной библиотеке Python и к их исходному программному коду. Работа с Юникодом может оказаться очень тонким делом.

Удаление файлов с байт-кодом

Модуль find из предыдущего раздела – не самый универсальный инструмент поиска строк из тех, что мы увидим далее, но несомненно является первым важным шагом. Он отбирает файлы, по которым затем можно реализовать поиск в сценарии автоматизации. Фактически наличие одной только операции отбора файлов из дерева каталогов достаточно для решения разнообразных задач системного администрирования.

Например, одна из типичных задач, которую мне приходится выполнять регулярно, заключается в удалении всех файлов с байт-кодом в дереве каталогов. Поскольку эти файлы не всегда являются переносимыми между основными версиями Python, их желательно удалять при подготовке программы к передаче пользователям, чтобы дать возможность интерпретатору самому создать новые файлы при первой же попытке импортировать модули. Теперь, когда мы получили богатый опыт использования функции os.walk, мы могли бы отказаться от услуг посредников и использовать ее непосредственно. В примере 6.14 представлен переносимый и универсальный инструмент командной строки, принимающий аргументы, обрабатывающий исключения и поддерживающий режим трассировки и простого поиска без удаления.

Пример 6.14. PP4E\Tools\cleanpyc.py

```
"""
удаляет все файлы .pyc с байт-кодом в дереве каталогов: аргумент командной
строки, если он указан, интерпретируется как корневой каталог, в противном
случае корневым считается текущий рабочий каталог
"""
import os, sys
findonly = False
rootdir = os.getcwd() if len(sys.argv) == 1 else sys.argv[1]

found = removed = 0
for (thisDirLevel, subsHere, filesHere) in os.walk(rootdir):
    for filename in filesHere:
        if filename.endswith('.pyc'):
            fullname = os.path.join(thisDirLevel, filename)
            print('=>', fullname)
            if not findonly:
                try:
```

```
                    os.remove(fullname)
                    removed += 1
                except:
                    type, inst = sys.exc_info()[:2]
                    print('*'*4, 'Failed:', filename, type, inst)
            found += 1

print('Found', found, 'files, removed', removed)
```

Если запустить этот сценарий, он выполнит обход дерева каталогов (CWD – по умолчанию, или дерева с корнем в каталоге, переданном в виде аргумента командной строки) и удалит все встретившиеся файлы с байт-кодом:

```
C:\...\Examples\PP4E> Tools\cleanpyc.py
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\__init__.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\initdata.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\make_db_file.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\manager.pyc
=> C:\Users\mark\Stuff\Books\4E\PP4E\dev\Examples\PP4E\Preview\person.pyc
...множество строк опущено...
Found 24 files, removed 24

C:\...\PP4E\Tools> cleanpyc.py .
=> .\find.pyc
=> .\visitor.pyc
=> .\__init__.pyc
Found 3 files, removed 3
```

Этот сценарий действует, но в нем выполняется чуть больше ручной работы, чем требуется. Теперь, когда мы знаем, как выполнять операцию поиска, создание сценариев на основе этих знаний становится практически тривиальным делом, которое сводится к поиску соответствующих имен файлов. Так, в примере 6.15 мы вновь используем команды оболочки, если они присутствуют в системе.

Пример 6.15. PP4E\Tools\cleanpyc-find-shell.py

```
"""
отыскивает и удаляет все файлы "*.pyc" с байт-кодом в дереве каталогов, имя
которого передается в виде аргумента командной строки; предполагает наличие
непереносимой Unix-подобной команды find
"""

import os, sys

rundir = sys.argv[1]
if sys.platform[:3] == 'win':
    findcmd = r'c:\cygwin\bin\find %s -name "*.pyc" -print' % rundir
else:
    findcmd = 'find %s -name "*.pyc" -print' % rundir
```

```
print(findcmd)

count = 0
for fileline in os.popen(findcmd):      # обход всех строк результата,
    count += 1                          # завершающихся символом \n
    print(fileline, end='')
    os.remove(fileline.rstrip())

print('Removed %d .pyc files' % count)
```

Этот сценарий удалит все файлы, имена которых возвращает команда оболочки:

```
C:\...\PP4E\Tools> cleanpyc-find-shell.py .
c:\cygwin\bin\find . -name "*.pyc" -print
./find.pyc
./visitor.pyc
./__init__.pyc
Removed 3 .pyc files
```

В этом сценарии функция `os.popen` получает вывод программы `find` из оболочки Cygwin, установленной на одном из моих Windows-компьютеров, или от стандартной команды `find`, имеющейся в Linux. Он также *абсолютно непереносим* на компьютеры, работающие под управлением Windows, если на них не установлена Unix-подобная программа `find`, а ее нет ни на одном из моих личных компьютеров (не говоря уже о большинстве компьютеров в мире в целом). Кроме того, как мы уже знаем, запуск команд оболочки из сценариев наносит ущерб производительности, поскольку при этом приходится запускать новую независимую программу.

Мы можем значительно улучшить переносимость и производительность и при этом сохранить программный код простым, применив инструмент поиска, написанный нами на языке Python в предыдущем разделе. Новый сценарий приводится в примере 6.16.

Пример 6.16. PP4E\Tools\cleanpyc-find-py.py

```
"""
отыскивает и удаляет все файлы "*.pyc" с байт-кодом в дереве каталогов, имя
которого передается в виде аргумента командной строки;
использует утилиту find, написанную на языке Python, за счет чего обеспечивается
переносимость;
запустите этот сценарий, чтобы удалить файлы .pyc, скомпилированные старой
версией Python;
"""

import os, sys, find # here, gets Tools.find

count = 0
for filename in find.find('*.pyc', sys.argv[1]):
    count += 1
```

```
        print(filename)
        os.remove(filename)

    print('Removed %d .pyc files' % count)
```

Как и прежде, этот сценарий удалит все файлы с байт-кодом в дереве каталогов с корнем, переданным в аргументе командной строки, но на этот раз наш сценарий может выполняться в любой системе, где установлен Python:

```
C:\...\PP4E\Tools> cleanpyc-find-py.py .
.\find.pyc
.\visitor.pyc
.\__init__.pyc
Removed 3 .pyc files
```

Этот сценарий переносим и позволяет избежать издержек, связанных с запуском внешних программ. Однако утилита `find` просто осуществляет поиск в дереве каталогов – она отбирает имена файлов, соответствующие шаблону, но не обращает внимания на их содержимое. Мы можем запрограммировать дополнительный поиск, используя результаты, полученные с помощью `find`, однако ручное управление позволит нам использовать процесс поиска более непосредственно. Как это можно сделать, демонстрируется в следующем разделе.

Сценарий Python для поиска в дереве

Наконец, после экспериментов с инструментами `grep`, `glob` и `find` для упрощения глобального поиска на всех платформах, которые могут мне когда-либо встретиться, я написал сценарий на языке Python, который выполняет основную работу вместо меня. В примере 6.17 применяются стандартные средства Python, с которыми мы познакомились в предыдущих главах: `os.walk` – для обхода файлов в каталоге, `os.path.splitext` – для пропуска файлов с расширениями, характерными для двоичных файлов, и `os.path.join` – для переносимого объединения путей к каталогам с именами файлов.

Поскольку он написан исключительно на языке Python, этот сценарий в равной степени может использоваться и в Linux, и в 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 использовании приема запуска команды оболочки.

Пример 6.17. PP4E\Tools\search_all.py

```
"""
###############################################################################
Порядок использования: "python ...\Tools\search_all.py dir string".
Отыскивает все файлы в указанном дереве каталогов, содержащие заданную строку;
для предварительного отбора имен файлов использует интерфейс os.walk вместо
```

```
find.find; вызывает visitfile для каждой строки в результатах, полученных
вызовом функции find.find с шаблоном "*";
###############################################################################
"""

import os, sys
listonly = False
textexts = ['.py', '.pyw', '.txt', '.c', '.h']   # игнорировать двоичные файлы

def searcher(startdir, searchkey):
    global fcount, vcount
    fcount = vcount = 0
    for (thisDir, dirsHere, filesHere) in os.walk(startdir):
        for fname in filesHere:                  # для каждого некаталога
            fpath = os.path.join(thisDir, fname) # fname не содержит пути
            visitfile(fpath, searchkey)

def visitfile(fpath, searchkey):                 # для каждого некаталога
    global fcount, vcount                        # искать строку
    print(vcount+1, '=>', fpath)                 # пропустить защищенные файлы
    try:
        if not listonly:
            if os.path.splitext(fpath)[1] not in textexts:
                print('Skipping', fpath)
            elif searchkey in open(fpath).read():
                input('%s has %s' % (fpath, searchkey))
                fcount += 1
    except:
        print('Failed:', fpath, sys.exc_info()[0])
    vcount += 1

if __name__ == '__main__':
    searcher(sys.argv[1], sys.argv[2])
    print('Found in %d files, visited %d' % (fcount, vcount))
```

Функционально этот сценарий делает примерно то, что мы получили бы, вызвав его функцию `visitfile` для всех строк, сгенерированных нашей функцией `find.find` с шаблоном «*». Но поскольку эта версия настроена на поиск по содержимому файлов, она лучше соответствует своей цели. В действительности это сходство обусловлено лишь использованием шаблона «*», который вынуждает `find.find` выполнить обход всех файлов, а это, собственно, то, чем занята новая функция `searcher`. Инструмент поиска хорошо подходит для выбора файлов определенного типа, при этом преимущество данного сценария состоит в возможности произвести определенные действия непосредственно в процессе обхода.

При запуске в виде самостоятельного сценария ключ поиска передается в командной строке, а при импортировании клиент вызывает функцию `searcher` непосредственно. Например, чтобы найти все вхождения строки в дереве примеров для книги, выполните в команду в оболочке DOS или Unix, как показано ниже:

```
C:\\PP4E> Tools\search_all.py . mimetypes
1 => .\LaunchBrowser.py
2 => .\Launcher.py
3 => .\Launch_PyDemos.pyw
4 => .\Launch_PyGadgets_bar.pyw
5 => .\__init__.py
6 => .\__init__.pyc
Skipping .\__init__.pyc
7 => .\Preview\attachgui.py
8 => .\Preview\bob.pkl
Skipping .\Preview\bob.pkl
```

...множество строк опущено: ожидает нажатия клавиши Enter после обнаружения каждого совпадения...

```
Found in 2 files, visited 184
```

Сценарий выводит список всех проверяемых им файлов, сообщает о пропущенных файлах (имена с расширениями, отсутствующими в переменной `textexts`, которые, как предполагается, являются двоичными файлами) и останавливается, ожидая нажатия клавиши Enter после вывода сообщения о нахождении в файле искомой строки. Точно так же сценарий `search_all` работает и при *импортировании*, но не выводит итоговой строки со статистикой (функции `fcount` и `vcount` находятся в модуле, и их также можно импортировать, чтобы получить итоговые сведения):

```
C:\...\PP4E\dev\Examples\PP4E> python
>>> import Tools.search_all
>>> search_all.searcher(r'C:\temp\PP3E\Examples', 'mimetypes')
```

... множество строк опущено: останавливается 8 раз в ожидании нажатия клавиши Enter...

```
>>> search_all.fcount, search_all.vcount      # совпадений, файлов
(8, 1429)
```

Каким бы образом ни запускался этот сценарий, он находит все вхождения искомой строки в целом дереве каталогов – например, изменившееся имя файла примера, объекта или каталога. Это именно то, что мне было необходимо, – по крайней мере, я так думал, пока не погрузился в дальнейшие размышления, пытаясь отыскать более полные и лучше структурированные решения, о которых рассказывается в следующем разделе.

Обязательно ознакомьтесь с обсуждением регулярных выражений в главе 19. В данном случае сценарий `search_all` пытается отыскать в каждом файле простую строку, применяя простое выражение проверки на вхождение, однако его легко можно было усовершенствовать, добавив возможность поиска по регулярному выражению (грубо говоря, для этого достаточно заменить оператор `in` вызовом метода поиска объекта регулярного выражения). Разумеется, это усовершенствование покажется вам намного проще, когда мы узнаем, как это делается.

Обратите также внимание на список `textexts` в примере 6.17. В нем перечислены все допустимые расширения текстовых файлов. Решение можно сделать более универсальным и надежным, если использовать логику модуля `mimetypes`, с которым мы познакомимся ближе к концу этой главы, который позволяет делать предположения о типе содержимого файла по его имени. Однако список допустимых расширений обеспечивает более точное управление, и его вполне достаточно для деревьев, поиск по которым я осуществлял с помощью этого сценария.

Наконец, обратите внимание, что для простоты во всех примерах поиска по дереву каталогов в этой главе предполагается, что текстовые файлы содержат текст Юникода, закодированный с применением кодировки по умолчанию, используемой платформой. Чтобы избежать ошибок при декодировании, в примерах можно было бы открывать текстовые файлы в двоичном режиме, но при этом результаты поиска могут оказаться неточными, если сравниваемые строки байтов будут закодированы с применением различных схем кодирования. Более удачное решение вы найдете в реализации утилиты «grep» в примере приложения PyEdit с графическим интерфейсом, где обеспечивается возможность применения указанной пользователем кодировки и пропускаются те текстовые или двоичные файлы, попытка декодирования которых завершается неудачей.

Visitor: обход каталогов «++»

Лень – двигатель прогресса. Имея в своем распоряжении переносимый сценарий `search_all` из примера 6.17, я мог точнее находить файлы, которые было необходимо отредактировать при изменении содержимого или структуры дерева примеров в книге. Первоначально 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 запуска редактора – занятие невеселое, особенно если нужно отредактировать много файлов. Поскольку у меня всегда найдется более интересное занятие, чем десятки раз запускать редактор вручную, я стал думать, как *автоматически* запускать редактор для каждого подозрительного файла.

К сожалению, сценарий `search_all` просто выводит полученные результаты на экран. Хотя этот текст можно перехватить и проанализировать с помощью другой программы, запускаемой функцией `os.popen`, проще может оказаться подход, когда редактор запускается прямо во время поиска, но для этого могут потребоваться большие изменения в реализации сценария. Здесь мне пришли в голову три мысли.

Избыточность

Написав несколько утилит обхода каталогов, я понял, что каждый раз я снова и снова пишу однотипный программный код. Обход можно упростить еще больше, если скрыть детали под оболочкой и тем самым упростить повторное использование решения. Инструмент `os.walk` позволяет избежать необходимости писать рекурсивные функции, но при его использовании выполняются лишние действия (например, присоединение имен каталогов, вывод трассировочной информации).

Расширяемость

Исходя из прошлого опыта, очевидно, что в долгосрочной перспективе легче добавлять новые возможности в универсальный механизм поиска в каталогах в виде внешних компонентов, чем менять программный код исходного сценария. Редактирование файлов могло быть одним из возможных расширений (а что вы думаете об автоматизации операции замены текста?), поэтому предпочтительнее выглядит более обобщенное и настраиваемое решение, допускающее возможность многократного использования. Функция `os.walk` достаточно проста в использовании, но прием, основанный на циклах, не так хорошо поддается настройке, как использование классов.

Инкапсуляция

Опираясь на прошлый опыт, я также знаю, что всегда желательно стараться максимально скрывать детали реализации инструментов от программ. Функция `os.walk` скрывает свою рекурсивную природу, тем не менее она предлагает весьма специфический интерфейс, который вполне может измениться в будущем. Подобные изменения имели место в прошлом – ближе к концу этого раздела я расскажу, как из версии Python 3.X был исключен один из инструментов обхода деревьев, что сразу же привело к нарушениям в работе программного кода, использующего его. Было бы лучше скрыть подобные зависимости за более нейтральным интерфейсом, чтобы клиентский программный код не приходил в негодность, как только нам потребуется внести изменения в реализацию нашего инструмента.

Конечно, если вы в достаточной мере изучили язык Python, то вы не можете не понимать, что все эти цели указывают на необходимость использования *объектно-ориентированного подхода* к реализации обхода и поиска. В примере 6.18 приводится одна из возможных реализаций этих целей. Этот модуль экспортирует универсальный класс `FileVisitor`, который в основном служит лишь оболочкой для `os.walk`, облегчающей использование и расширение, а также базовый класс `SearchVisitor`, обобщающий идею поиска в каталоге.

Сам по себе класс `SearchVisitor` делает то же самое, что делал сценарий `search_all`, но кроме этого, он открывает новые возможности по на-

стройке процедуры поиска – 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: достаточно просто определить подкласс, в котором будут добавлены специфические для поиска расширения. То же относится и к классу `FileVisitor` – переопределяя его методы и используя его атрибуты, можно внедряться в процесс обхода деревьев, используя приемы ООП. Это обычное дело в программировании – как только вы начинаете достаточно часто решать одни и те же тактические задачи, они наталкивают вас на подобные стратегические размышления.

Пример 6.18. PP4E\Tools\visitor.py

```
"""
###############################################################################
Тест: "python ...\Tools\visitor.py dir testmask [строка]". Использует
классы и подклассы для сокрытия деталей использования функции os.walk при
обходе и поиске; testmask - битовая маска, каждый бит в которой определяет
тип самопроверки; смотрите также: подклассы visitor_*/.py; вообще подобные
фреймворки должны использовать псевдочастные имена вида __X, однако в данной
реализации все имена экспортируются для использования в подклассах и клиентами;
переопределите метод reset для поддержки множественных, независимых объектов-
обходчиков, требующих обновлений в подклассах;
###############################################################################
"""

import os, sys

class FileVisitor:
    """
    Выполняет обход всех файлов, не являющихся каталогами, ниже startDir
    (по умолчанию '.'); при создании собственных обработчиков
    файлов/каталогов переопределяйте методы visit*; аргумент/атрибут context
    является необязательным и предназначен для хранения информации,
    специфической для подкласса; переключатель режима трассировки trace: 0 -
    нет трассировки, 1 - подкаталоги, 2 - добавляются файлы
    """
    def __init__(self, context=None, trace=2):
        self.fcount  = 0
        self.dcount  = 0
        self.context = context
        self.trace   = trace

    def run(self, startDir=os.curdir, reset=True):
        if reset: self.reset()
        for (thisDir, dirsHere, filesHere) in os.walk(startDir):
            self.visitdir(thisDir)
            for fname in filesHere:                  # для некаталогов
                fpath = os.path.join(thisDir, fname) # fname не содержит пути
                self.visitfile(fpath)
```

```
    def reset(self):                          # используется обходчиками,
        self.fcount = self.dcount = 0         # выполняющими обход независимо

    def visitdir(self, dirpath):              # вызывается для каждого каталога
        self.dcount += 1                      # переопределить или расширить
        if self.trace > 0: print(dirpath, '...')

    def visitfile(self, filepath):            # вызывается для каждого файла
        self.fcount += 1                      # переопределить или расширить
        if self.trace > 1: print(self.fcount, '=>', filepath)

class SearchVisitor(FileVisitor):
    """
    Выполняет поиск строки в файлах, находящихся в каталоге startDir и ниже;
    в подклассах: переопределите метод visitmatch, списки расширений, метод
    candidate, если необходимо; подклассы могут использовать testexts, чтобы
    определить типы файлов, в которых может выполняться поиск (но могут также
    переопределить метод candidate, чтобы использовать модуль mimetypes для
    определения файлов с текстовым содержимым: смотрите далее)
    """

    skipexts = []
    testexts = ['.txt', '.py', '.pyw', '.html', '.c', '.h'] # допустимые расш.
   #skipexts = ['.gif', '.jpg', '.pyc', '.o', '.a', '.exe'] # или недопустимые
                                                            # расширения
    def __init__(self, searchkey, trace=2):
        FileVisitor.__init__(self, searchkey, trace)
        self.scount = 0

    def reset(self):                          # в независимых обходчиках
        self.scount = 0

    def candidate(self, fname):               # переопределить, если желательно
        ext = os.path.splitext(fname)[1]      # использовать модуль mimetypes
        if self.testexts:
            return ext in self.testexts       # если допустимое расширение
        else:                                 # или, если недопустимое
            return ext not in self.skipexts   # расширение

    def visitfile(self, fname):               # поиск строки
        FileVisitor.visitfile(self, fname)
        if not self.candidate(fname):
            if self.trace > 0: print('Skipping', fname)
        else:
            text = open(fname).read()         # 'rb' для недекодируемого текста
            if self.context in text:          # или text.find() != -1
                self.visitmatch(fname, text)
                self.scount += 1

    def visitmatch(self, fname, text):              # обработка совпадения
        print('%s has %s' % (fname, self.context)) # переопределить
```

```
if __name__ == '__main__':
    # логика самотестирования
    dolist   = 1
    dosearch = 2                              # 3 = список и поиск
    donext   = 4                              # при добавлении следующего теста

    def selftest(testmask):
        if testmask & dolist:
            visitor = FileVisitor(trace=2)
            visitor.run(sys.argv[2])
            print('Visited %d files and %d dirs' %
                  (visitor.fcount, visitor.dcount))

        if testmask & dosearch:
            visitor = SearchVisitor(sys.argv[3], trace=0)
            visitor.run(sys.argv[2])
            print('Found in %d files, visited %d' %
                  (visitor.scount, visitor.fcount))

selftest(int(sys.argv[1]))                    # например, 3 = dolist | dosearch
```

Этот модуль служит в основном для экспорта классов, используемых другими программами, но и при запуске в виде самостоятельного сценария 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:

```
C:\...\PP4E\Tools> visitor.py 1 C:\temp\PP3E\Examples
C:\temp\PP3E\Examples ...
1 => C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E ...
2 => C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
3 => C:\temp\PP3E\Examples\PP3E\LaunchBrowser.pyw
4 => C:\temp\PP3E\Examples\PP3E\Launcher.py
5 => C:\temp\PP3E\Examples\PP3E\Launcher.pyc
...множество строк опущено (передайте по конвейеру команде more или перенаправьте в файл)...
1424 => C:\temp\PP3E\Examples\PP3E\System\Threads\thread-count.py
1425 => C:\temp\PP3E\Examples\PP3E\System\Threads\thread1.py
C:\temp\PP3E\Examples\PP3E\TempParts ...
1426 => C:\temp\PP3E\Examples\PP3E\TempParts\109_0237.JPG
1427 => C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
1428 => C:\temp\PP3E\Examples\PP3E\TempParts\part-001.txt
1429 => C:\temp\PP3E\Examples\PP3E\TempParts\part-002.html
Visited 1429 files and 186 dirs
```

Если же вызвать этот сценарий с 2 в первом аргументе, он создаст и запустит объект SearchVisitor, используя третий аргумент в качестве ключа поиска. Эта форма напоминает запуск знакомого нам сценария

search_all.py, но в данном случае при обнаружении совпадений сценарий не останавливается:

```
C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples mimetypes
C:\temp\PP3E\Examples\PP3E\extras\LosAlamosAdvancedClass\day1-system\data.txt
has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py has
mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py has mimetypes
Found in 8 files, visited 1429
```

Технически при передаче сценарию числа 3 в первом аргументе он выполнит *оба* объекта, FileVisitor и SearchVisitor (осуществив два отдельных обхода). Первый аргумент в действительности используется в качестве битовой маски для выбора одной или более поддерживаемых самопроверок – если бит для какого-либо теста установлен в двоичном значении аргумента, этот тест будет выполнен. Поскольку 3 представляется в двоичном виде, как 011, выбираются одновременно поиск (010) и вывод списка (001). В более дружественной системе можно было бы определить символические параметры (например, искать аргументы -search и -list), но для целей данного сценария достаточно битовых масок.

Как обычно, этот модуль можно также использовать в интерактивном сеансе. Ниже приводится один из способов определения количества файлов и каталогов внутри определенного каталога. Последняя команда выполняет обход всего жесткого диска (и выводит результаты после заметной задержки!). Смотрите также пример «Найди самый большой файл Python» в начале этой главы, где описываются такие проблемы, как повторное посещение подкаталогов, не обрабатываемые данной реализацией:

```
C:\...\PP4E\Tools> python
>>> from visitor import FileVisitor
>>> V = FileVisitor(trace=0)
>>> V.run(r'C:\temp\PP3E\Examples')
>>> V.dcount, V.fcount
(186, 1429)

>>> V.run('..')              # независимый обход (сброс счетчиков)
>>> V.dcount, V.fcount
(19, 181)

>>> V.run('..', reset=False) # накопительный обход (счетчики сохраняются)
>>> V.dcount, V.fcount
(38, 362)
```

```
>>> V = FileVisitor(trace=0) # новый независимый обходчик (свои счетчики)
>>> V.run(r'C:\\')           # весь диск: в Unix попробуйте '/'
>>> V.dcount, V.fcount
(24992, 198585)
```

Модуль visitor удобно использовать как самостоятельный сценарий, чтобы получить список файлов и выполнить поиск в дереве каталогов, но в действительности он создавался, чтобы служить основой для расширения. В оставшейся части этого раздела мы коротко познакомимся с некоторыми клиентами этого модуля, которые добавляют свои операции с деревьями каталогов, используя приемы ООП.

Редактирование файлов в деревьях каталогов (Visitor)

Теперь, после обобщения обхода деревьев и поиска, легко сделать следующий шаг и добавить отдельный, совершенно новый компонент автоматического редактирования файлов. В примере 6.19 приводится определение нового класса EditVisitor, который просто переопределяет метод visitmatch класса SearchVisitor, новая версия которого открывает 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 приобретается по наследству.

Пример 6.19. PP4E\Tools\visitor_edit.py

```
"""
Порядок использования: "python ...\Tools\visitor_edit.py string rootdir?".
Добавляет подкласс класса SearchVisitor, который автоматически запускает
текстовый редактор. В процессе обхода автоматически открывает в текстовом
редакторе файлы, содержащие искомую строку; в Windows можно также использовать
editor='edit' или 'notepad'; чтобы воспользоваться текстовым редактором,
реализация которого будет представлена далее в книге, попробуйте r'python Gui\
TextEditor\textEditor.py'; при работе с некоторыми редакторами можно было бы
передать команду перехода к первому совпадению с искомой строкой;
"""

import os, sys
from visitor import SearchVisitor

class EditVisitor(SearchVisitor):
    """
    открывает для редактирования файлы, содержащие искомую строку и
    находящиеся в каталоге startDir и ниже
    """
    editor = r'C:\cygwin\bin\vim-nox.exe' # у вас может быть другой редактор!

    def visitmatch(self, fpathname, text):
        os.system('%s %s' % (self.editor, fpathname))
```

```
if __name__ == '__main__':
    visitor = EditVisitor(sys.argv[1])
    visitor.run('.' if len(sys.argv) < 3 else sys.argv[2])
    print('Edited %d files, visited %d' % (visitor.scount, visitor.fcount))
```

При использовании объекта `EditVisitor` текстовый редактор запускается посредством передачи командной строки функции `os.system`, которая обычно блокирует вызывающий программный код до момента, когда завершится порожденная программа. На моих машинах при каждом обнаружении сценарием совпадения во время обхода запускается текстовый редактор vi в том окне консоли, где был запущен сценарий. При выходе из редактора обход дерева возобновляется.

Найдем и отредактируем несколько файлов. При запуске этого файла как самостоятельного сценария мы передаем ему искомую строку в аргументе командной строки (здесь используется строка «mimetypes»). Корневой каталог всегда передается методу `run` как «.» (текущий рабочий каталог). Сообщения о состоянии обхода выводятся на консоль, но каждый файл, в котором обнаружено совпадение с искомой строкой, тут же автоматически открывается в текстовом редакторе. В данном случае редактор запускается восемь раз – попробуйте запустить этот сценарий в своем дереве каталогов и со своим редактором, чтобы лучше почувствовать, как он работает:

```
C:\...\PP4E\Tools> visitor_edit.py mimetypes C:\temp\PP3E\Examples
C:\temp\PP3E\Examples ...
1 => C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E ...
2 => C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
3 => C:\temp\PP3E\Examples\PP3E\LaunchBrowser.pyw
4 => C:\temp\PP3E\Examples\PP3E\Launcher.py
5 => C:\temp\PP3E\Examples\PP3E\Launcher.pyc
Skipping C:\temp\PP3E\Examples\PP3E\Launcher.pyc
```

...множество строк опущено...

```
1427 => C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
Skipping C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
1428 => C:\temp\PP3E\Examples\PP3E\TempParts\part-001.txt
1429 => C:\temp\PP3E\Examples\PP3E\TempParts\part-002.html
Edited 8 files, visited 1429
```

В итоге получился именно тот инструмент, который мне был нужен, чтобы упростить сопровождение дерева примеров книги. После значительных изменений, например задания имен совместно используемых модулей или файлов и каталогов, я запускаю этот сценарий в корневом каталоге дерева примеров с соответствующей строкой поиска и нужным образом редактирую все открывающиеся файлы. Мне все еще приходится вручную изменять файлы в редакторе, но это зачастую безопаснее, чем вслепую выполнять глобальную замену.

Глобальная замена в деревьях каталогов (Visitor)

Но раз уж я затронул этот вопрос, то, имея общий класс для обхода дерева, легко написать и подкласс для глобального поиска и замены. В примере 6.20 приводится определение класса `ReplaceVisitor`, наследующего класс `FileVisitor`, который переопределяет 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 изменившихся файлов, чтобы их можно было просмотреть и проверить автоматически сделанные изменения (можно, например, автоматически вызывать текстовый редактор для каждого измененного файла).

Пример 6.20. PP4E\Tools\visitor_replace.py

```
"""
Использование: "python ...\Tools\visitor_replace.py rootdir fromStr toStr".
Выполняет глобальный поиск с заменой во всех файлах в дереве каталогов: заменяет
fromStr на toStr во всех текстовых файлах; это мощный, но опасный инструмент!!
visitor_edit.py запускает редактор, чтобы дать возможность проверить и внести
коррективы, и поэтому он более безопасный; чтобы просто получить список
соответствующих файлов, используйте visitor_collect.py; режим простого вывода
списка здесь напоминает SearchVisitor и CollectVisitor;
"""

import sys
from visitor import SearchVisitor

class ReplaceVisitor(SearchVisitor):
    """
    Заменяет fromStr на toStr в файлах в каталоге startDir и ниже;
    имена изменившихся файлов сохраняются в списке obj.changed
    """
    def __init__(self, fromStr, toStr, listOnly=False, trace=0):
        self.changed  = []
        self.toStr    = toStr
        self.listOnly = listOnly
        SearchVisitor.__init__(self, fromStr, trace)

    def visitmatch(self, fname, text):
        self.changed.append(fname)
        if not self.listOnly:
            fromStr, toStr = self.context, self.toStr
            text = text.replace(fromStr, toStr)
            open(fname, 'w').write(text)

if __name__ == '__main__':
    listonly = input('List only?') == 'y'
    visitor = ReplaceVisitor(sys.argv[2], sys.argv[3], listonly)
    if listonly or input('Proceed with changes?') == 'y':
        visitor.run(startDir=sys.argv[1])
```

```
    action = 'Changed' if not listonly else 'Found'
    print('Visited %d files' % visitor.fcount)
    print(action, '%d files:' % len(visitor.changed))
    for fname in visitor.changed: print(fname)
```

Чтобы применить этот сценарий к определенному дереву каталогов, выполните команду, как показано ниже, указав соответствующую искомую строку и строку замены. На моем, жутко нерасторопном нетбуке, обработка дерева с 1429 файлами, из которых 101 потребовалось изменить, заняла примерно три секунды реального времени, когда система была не слишком занята другими задачами:

```
C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?y
Visited 1429 files
Found 101 files:
C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
C:\temp\PP3E\Examples\PP3E\Launcher.py
```

...большое количество имен файлов, соответствующих критерию поиска, опущено...

```
C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?n
Proceed with changes?y
Visited 1429 files
Changed 101 files:
C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
C:\temp\PP3E\Examples\PP3E\Launcher.py
```

...большое количество имен изменившихся файлов опущено...

```
C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?n
Proceed with changes?y
Visited 1429 files
Changed 0 files:
```

Естественно, проверить работу этого сценария можно с помощью сценария `visitor` (и суперкласса `SearchVisitor`):

```
C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples PP3E
Found in 0 files, visited 1429

C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples PP4E
C:\temp\PP3E\Examples\README-root.txt has PP4E
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw has PP4E
C:\temp\PP3E\Examples\PP3E\Launcher.py has PP4E
```

...большое количество имен файлов, соответствующих критерию поиска, опущено...

```
Found in 101 files, visited 1429
```

Это одновременно очень мощный и опасный сценарий. 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 поможет избежать необходимости вручную редактировать подозрительные файлы. Например, адреса веб-сайтов в файлах HTML достаточно специфичны, чтобы случайно появиться в других местах.

Подсчет строк исходного программного кода (Visitor)

Два приведенных выше примера использования модуля visitor были ориентированы на выполнение поиска, однако базовый класс, реализующий обход дерева каталогов, легко можно было бы расширить для реализации более специфических задач. Так, в примере 6.21 приводится сценарий, расширяющий класс FileVisitor возможностью подсчета количества строк в файлах с исходными текстами программ во всем дереве каталогов. Принцип его действия основан на вызове метода visitfile этого класса для каждого файла, найденного инструментом поиска, написанным нами выше в этой главе, но применение ООП обеспечивает более высокую гибкость и расширяемость.

Пример 6.21. PP4E\Tools\visitor_sloc.py

```
"""
Подсчитывает строки во всех файлах с исходными текстами программ в дереве
каталогов, указанном в командной строке, и выводит сводную информацию,
сгруппированную по типам файлов (по расширениям). Реализует простейший алгоритм
SLOC (source lines of code - строки исходного текста): если необходимо, добавьте
пропуск пустых строк и комментариев.
"""

import sys, pprint, os
from visitor import FileVisitor

class LinesByType(FileVisitor):
    srcExts = [] # define in subclass

    def __init__(self, trace=1):
        FileVisitor.__init__(self, trace=trace)
        self.srcLines = self.srcFiles = 0
        self.extSums = {ext: dict(files=0, lines=0) for ext in self.srcExts}

    def visitsource(self, fpath, ext):
        if self.trace > 0: print(os.path.basename(fpath))
        lines = len(open(fpath, 'rb').readlines())
        self.srcFiles += 1
        self.srcLines += lines
        self.extSums[ext]['files'] += 1
        self.extSums[ext]['lines'] += lines
```

```
    def visitfile(self, filepath):
        FileVisitor.visitfile(self, filepath)
        for ext in self.srcExts:
            if filepath.endswith(ext):
                self.visitsource(filepath, ext)
                break

class PyLines(LinesByType):
    srcExts = ['.py', '.pyw'] # just python files

class SourceLines(LinesByType):
    srcExts = ['.py', '.pyw', '.cgi', '.html', '.c', '.cxx', '.h', '.i']

if __name__ == '__main__':
    walker = SourceLines()
    walker.run(sys.argv[1])
    print('Visited %d files and %d dirs' % (walker.fcount, walker.dcount))
    print('-'*80)
    print('Source files=>%d, lines=>%d' % (walker.srcFiles, walker.srcLines))
    print('By Types:')
    pprint.pprint(walker.extSums)
    print('\nCheck sums:', end=' ')
    print(sum(x['lines'] for x in walker.extSums.values()), end=' ')
    print(sum(x['files'] for x in walker.extSums.values()))
    print('\nPython only walk:')
    walker = PyLines(trace=0)
    walker.run(sys.argv[1])
    pprint.pprint(walker.extSums)
```

Если запустить его как самостоятельный сценарий, в процессе обхода будут выводиться трассировочные сообщения (опущены здесь для экономии места) и в конце будет представлен отчет о количестве строк, сгруппированный по типам файлов. Выполните этот сценарий в своем дереве каталогов, чтобы увидеть, как он действует. В моем дереве каталогов содержится 907 файлов с исходными текстами, насчитывающих 48 000 строк, включая 783 файла (*.py*) и 34 000 строк с исходными текстами на языке Python:

```
C:\...\PP4E\Tools> visitor_sloc.py C:\temp\PP3E\Examples
Visited 1429 files and 186 dirs
--------------------------------------------------------------------------------
Source files=>907, lines=>48047
By Types:
{'.c': {'files': 45, 'lines': 7370},
 '.cgi': {'files': 5, 'lines': 122},
 '.cxx': {'files': 4, 'lines': 2278},
 '.h': {'files': 7, 'lines': 297},
 '.html': {'files': 48, 'lines': 2830},
 '.i': {'files': 4, 'lines': 49},
```

```
 '.py': {'files': 783, 'lines': 34601},
 '.pyw': {'files': 11, 'lines': 500}}

Check sums: 48047 907

Python only walk:
{'.py': {'files': 783, 'lines': 34601}, '.pyw': {'files': 11, 'lines': 500}}
```

Копирование деревьев каталогов с помощью классов (Visitor)

Рассмотрим еще один случай использования классов-обходчиков. Когда я впервые написал сценарий `cpall.py` ранее в этой главе, я не понимал, как можно применить иерархию классов-обходчиков, с которой мы встретились выше, для копирования деревьев каталогов. При копировании необходимо выполнять обход сразу двух деревьев каталогов (оригинального и его копии), а `visitor` выполняет обход только одного дерева, используя функцию `os.walk`. Тогда мне казалось, что будет совсем не просто следить за тем, где находится сценарий в копии дерева.

Хитрость, до которой я в конце концов додумался, заключается в том, что следить за позицией в дереве копии вообще не требуется. Вместо этого сценарий в примере 6.22 просто замещает строку пути к каталогу «из» строкой пути к каталогу «в» и добавляет ее перед всеми именами каталогов, полученными от функции `os.walk`. В результате такой подмены получаются пути, куда должны быть скопированы оригинальные файлы и каталоги.

Пример 6.22. PP4E\Tools\visitor_cpall.py

```
"""
Использование: "python ...\Tools\visitor_cpall.py fromDir toDir trace?"
Действует подобно сценарию System\Filetools\cpall.py, но использует классы-
обходчики и функцию os.walk; заменяет строку fromDir на toDir перед всеми
именами, возвращаемыми обходчиком; предполагается, что изначально каталог toDir
не существует;
"""

import os
from visitor import FileVisitor                  # обходчик в каталоге '.'
from PP4E.System.Filetools.cpall import copyfile # PP4E - в пути поиска

class CpallVisitor(FileVisitor):
    def __init__(self, fromDir, toDir, trace=True):
        self.fromDirLen = len(fromDir) + 1
        self.toDir = toDir
        FileVisitor.__init__(self, trace=trace)

    def visitdir(self, dirpath):
        toPath = os.path.join(self.toDir, dirpath[self.fromDirLen:])
```

```
            if self.trace: print('d', dirpath, '=>', toPath)
            os.mkdir(toPath)
            self.dcount += 1

        def visitfile(self, filepath):
            toPath = os.path.join(self.toDir, filepath[self.fromDirLen:])
            if self.trace: print('f', filepath, '=>', toPath)
            copyfile(filepath, toPath)
            self.fcount += 1

    if __name__ == '__main__':
        import sys, time
        fromDir, toDir = sys.argv[1:3]
        trace = len(sys.argv) > 3
        print('Copying...')
        start = time.clock()
        walker = CpallVisitor(fromDir, toDir, trace)
        walker.run(startDir=fromDir)
        print('Copied', walker.fcount, 'files,', walker.dcount, 'directories',
              end=' ')
        print('in', time.clock() - start, 'seconds')
```

Эта версия достигает почти той же цели, что и оригинал, но в ней сделано несколько допущений, чтобы упростить реализацию. Предполагается, что каталог «в» изначально не существует, а исключения, возникающие в процессе копирования, не игнорируются. Следующий пример демонстрирует копирование дерева примеров для предыдущего издания книги в Windows:

```
C:\...\PP4E\Tools> set PYTHONPATH
PYTHONPATH=C:\Users\Mark\Stuff\Books\4E\PP4E\dev\Examples

C:\...\PP4E\Tools> rmdir /S copytemp
copytemp, Are you sure (Y/N)? y

C:\...\PP4E\Tools> visitor_cpall.py C:\temp\PP3E\Examples copytemp
Copying...
Copied 1429 files, 186 directories in 11.1722033777 seconds

C:\...\PP4E\Tools> fc /B copytemp\PP3E\Launcher.py
                       C:\temp\PP3E\Examples\PP3E\Launcher.py
Comparing files COPYTEMP\PP3E\Launcher.py and C:\TEMP\PP3E\EXAMPLES\PP3E\
LAUNCHER.PY
FC: no differences encountered
```

Несмотря на то, что в этой версии выполняется дополнительная операция извлечения среза из строки, она действует так же быстро, как и оригинал (фактическое отличие можно списать на разницу в нагрузке на систему). Кроме того, эта версия позволяет проследить за всеми путями «из» и «в» в процессе обхода, если передать ей третий аргумент командной строки:

```
C:\...\PP4E\Tools> rmdir /S copytemp
copytemp, Are you sure (Y/N)? y

C:\...\PP4E\Tools> visitor_cpall.py C:\temp\PP3E\Examples copytemp 1
Copying...
d C:\temp\PP3E\Examples => copytemp\
f C:\temp\PP3E\Examples\README-root.txt => copytemp\README-root.txt
d C:\temp\PP3E\Examples\PP3E => copytemp\PP3E
```

...множество строк опущено: запустите этот сценарий у себя, чтобы увидеть вывод целиком...

Другие примеры обходчиков (внешние)

Инструменту обхода деревьев на основе классов можно найти самые разные применения, однако у нас недостаточно места в книге, чтобы исследовать дополнительные подклассы. Дополнительные примеры реализации клиентов и их использования вы найдете в следующих примерах, входящих в состав пакета, описанного в Предисловии:

- *Tools\visitor_collect.py* отбирает и/или выводит имена файлов, содержащие искомую строку
- *Tools\visitor_poundbang.py* замещает пути к каталогам в строках «#!», находящихся в начале файлов сценариев в Unix
- *Tools\visitor_cleanpyc.py* – переработанная версия сценария удаления файлов с байт-кодом, использующая классы-обходчики
- *Tools\visitor_bigpy.py* – версия примера «Найди самый большой файл Python», приводившегося в начале главы, использующая классы-обходчики

Реализация большинства из них выглядит почти тривиально, как и реализация *visitor_edit.py* в примере 6.19, благодаря тому что все детали обхода деревьев каталогов автоматически скрываются за интерфейсом классов-обходчиков. Реализация отбора файлов, например, просто добавляет в список имена файлов, соответствующих критериям поиска, и позволяет переопределить список допустимых расширений имен файлов в каждом экземпляре – она напоминает комбинацию команд `find` и `grep` в Unix:

```
>>> from visitor_collect import CollectVisitor
>>> V = CollectVisitor('mimetypes', testexts=['.py', '.pyw'], trace=0)
>>> V.run(r'C:\temp\PP3E\Examples')
>>> for name in V.matches: print(name) # файлы .py и .pyw со строкой
...                                   # 'mimetypes'
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py
```

```
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py

C:\...\PP4E\Tools> visitor_collect.py mimetypes C:\temp\PP3E\Examples # как
                                                                    # сценарий
```

Основная логика сценария поиска наибольшего файла также выглядит достаточно просто и напоминает логику сценария в начале главы:

```
class BigPy(FileVisitor):
    def __init__(self, trace=0):
        FileVisitor.__init__(self, context=[], trace=trace)

    def visitfile(self, filepath):
        FileVisitor.visitfile(self, filepath)
        if filepath.endswith('.py'):
            self.context.append((os.path.getsize(filepath), filepath))
```

Пример реализации удаления файлов байт-кода на основе классов-обходчиков также возвращает нас назад, демонстрируя альтернативное решение задачи, с которой мы уже сталкивались выше в этой главе. Это, по сути, тот же самый программный код, но он вызывает функцию `os.remove` при обнаружении файлов с расширением «.рус».

Наконец, хотя классы-обходчики действительно являются всего лишь обертками вокруг функции `os.walk`, тем не менее они автоматизируют рутинные операции, выполняемые при обходе, и обеспечивают универсальный фундамент и альтернативную структуру на основе классов, которая может оказаться более естественной для некоторых решений, чем неструктурированные циклы. Кроме того, они наглядно демонстрируют, насколько хорошо поддержка ООП в языке Python позволяет отображать реальные структуры, такие как файловые системы. Приемы на основе функции `os.walk` хорошо подходят для одноразовых сценариев, тогда как лучшая расширяемость, уменьшенная избыточность и сокрытие, свойственные приемам ООП, могут быть основным вкладом в разработку действующих программ, которые могут развиваться в течение долгого времени с учетом наших потребностей.

Эти потребности *действительно* изменились с течением времени. Между третьим и четвертым изданиями этой книги из Python 3.X была исключена оригинальная функция `os.path.walk`, и функция `os.walk` стала единственным инструментом автоматизированного обхода деревьев в стандартной библиотеке. Это привело к тому, что примеры из предыдущего издания, использовавшие `os.path.walk`, оказались неработоспособными. Тогда как клиенты, реализованные на основе классов-обходчиков, использующих ту же функцию, не пострадали. Поскольку перевод классов-обходчиков на использование функции `os.walk` не отразился на их интерфейсе, инструменты, использующие их, сохранили работоспособность без переделки.

Это, пожалуй, один из самых ярких примеров преимуществ инкапсуляции в ООП. Будущее нельзя предсказать с достаточной степенью надежности, тем не менее на практике инструменты, определяемые пользователем, такие как классы-обходчики, обычно обеспечивают более полный контроль над изменениями в инструментах стандартной библиотеки, таких как `os.walk`. Можете поверить мне на слово – как человек, который обновлял три книги о языке Python на протяжении последних 15 лет, я могу с определенной долей уверенности сказать, что Python будет изменяться постоянно!

Проигрывание медиафайлов

В этой главе у нас еще осталось немного места, поэтому закроем ее небольшим развлекательным примером. Обратили ли вы внимание, что в сценариях поиска в двух предыдущих разделах мы жестко определяли расширения имен текстовых и двоичных файлов? Такой подход вполне оправдан для деревьев каталогов, к которым мы применяли их, но в общем случае его нельзя назвать законченным или переносимым. Было бы лучше, если бы мы могли автоматически определять тип файла по его имени. Такую возможность предоставляет модуль `mimetypes`. В этом разделе мы будем использовать его при создании сценария, который будет пытаться запускать файлы, опираясь на их типы, и по ходу дела разработаем универсальные инструменты для открытия медиафайлов переносимым способом, с использованием специализированных или универсальных программ-проигрывателей.

Как мы уже видели, в Windows эта задача решается тривиально просто – функция `os.startfile` открывает файл, используя реестр Windows, где хранятся соответствия между расширениями имен файлов и программами для работы с ними. На других платформах можно либо запускать определенные программы-проигрыватели для каждого типа файлов, либо возвращаться к использованию веб-броузера по умолчанию, открывая файлы с помощью модуля `webbrowser`. Воплощение этих идей в программный код приводится в примере 6.23.

Пример 6.23. PP4E\System\Media\playfile.py

```
#!/usr/local/bin/python
"""
################################################################################
Пытается проигрывать медиафайлы различных типов. Позволяет определять
специализированные программы-проигрыватели вместо использования универсального
приема открытия файла в веб-броузере. В текущем своем виде может не работать
в вашей системе; для открытия аудиофайлов в Unix используются фильтры и команды,
в Windows используется команда start, учитывающая ассоциации с расширениями
имен файлов (то есть для открытия файлов .au, например, она может запустить
проигрыватель аудиофайлов или веб-броузер). Настраивайте и расширяйте сценарий
```

```
под свои потребности. Функция playknownfile предполагает, что вы знаете, какой
тип медиафайла пытаетесь открыть, а функция playfile пробует определить тип
файла автоматически, используя модуль mimetypes; обе они пробуют запустить веб-
броузер с помощью модуля webbrowser, если тип файла не удается определить.
################################################################################
"""

import os, sys, mimetypes, webbrowser

helpmsg = """
Sorry: can't find a media player for '%s' on your system!
Add an entry for your system to the media player dictionary
for this type of file in playfile.py, or play the file manually.
"""

def trace(*args): print(*args)   # с разделяющими пробелами

################################################################################
# приемы проигрывания: универсальный и другие: дополните своими приемами
################################################################################

class MediaTool:
    def __init__(self, runtext=''):
        self.runtext = runtext
    def run(self, mediafile, **options):         # options обычно игнорируется
        fullpath = os.path.abspath(mediafile)    # cwd может быть любым
        self.open(fullpath, **options)

class Filter(MediaTool):
    def open(self, mediafile, **ignored):
        media  = open(mediafile, 'rb')
        player = os.popen(self.runtext, 'w')     # запустить команду оболочки
        player.write(media.read())               # отправить файл в stdin

class Cmdline(MediaTool):
    def open(self, mediafile, **ignored):
        cmdline = self.runtext % mediafile   # запустить команду
        os.system(cmdline)                   # использовать %s для имени файла

class Winstart(MediaTool):                           # использует реестр Windows
    def open(self, mediafile, wait=False, **other): # позволяет дождаться
        if not wait:                                 # окончания проигрывания файла
            os.startfile(mediafile)                  # или os.system('start file')
        else:
            os.system('start /WAIT ' + mediafile)

class Webbrowser(MediaTool):
    # file:// требует указывать абсолютный путь
    def open(self, mediafile, **options):
        webbrowser.open_new('file://%s' % mediafile, **options)
```

```
##########################################################################
# медиа- и платформозависимые методы: измените или укажите один из имеющихся
##########################################################################

# соответствия платформ и проигрывателей: измените!

audiotools = {
    'sunos5': Filter('/usr/bin/audioplay'),    # os.popen().write()
    'linux2': Cmdline('cat %s > /dev/audio'),  # по крайней мере в PDA Zaurus
    'sunos4': Filter('/usr/demo/SOUND/play'),  # да, даже такая древность!
    'win32':  Winstart()                       # startfile или system
   #'win32':  Cmdline('start %s')
}

videotools = {
    'linux2': Cmdline('tkcVideo_c700 %s'),     # PDA Zaurus
    'win32':  Winstart(),                      # предотвратить вывод окна DOS
}

imagetools = {
    'linux2': Cmdline('zimager %s'),           # PDA Zaurus
    'win32':  Winstart(),
}

texttools = {
    'linux2': Cmdline('vi %s'),                # PDA Zaurus
    'win32':  Cmdline('notepad %s')            # или попробовать PyEdit?
}

apptools = {
    'win32': Winstart()                        # doc, xls, и др.: используйте
                                               # на свой страх и риск!
}

# таблица соответствия между типами файлов и программами-проигрывателями

mimetable = {'audio':       audiotools,
             'video':       videotools,
             'image':       imagetools,
             'text':        texttools,         # не-html текст: броузер
             'application': apptools}

##########################################################################
# интерфейсы высокого уровня
##########################################################################

def trywebbrowser(filename, helpmsg=helpmsg, **options):
    """
    пытается открыть файл в веб-броузере
    как последнее средство, если тип файла или платформы неизвестен,
    а также для файлов типа text/html
```

```
    """
    trace('trying browser', filename)
    try:
        player = Webbrowser()               # открыть в броузере
        player.run(filename, **options)
    except:
        print(helpmsg % filename)           # никакой из способов не работает

def playknownfile(filename, playertable={}, **options):
    """
    проигрывает медиафайл известного типа: использует программы-проигрыватели
    для данной платформы или запускает веб-броузер, если для этой платформы не
    определено ничего другого; принимает таблицу соответствий расширений и
    программ-проигрывателей
    """
    if sys.platform in playertable:                         # известный
        playertable[sys.platform].run(filename, **options) # инструмент
    else:                                                   # универсальный
        trywebbrowser(filename, **options)                  # прием

def playfile(filename, mimetable=mimetable, **options):
    """
    проигрывает медиафайл любого типа: использует модуль mimetypes для
    определения типа медиафайла и таблицу соответствий между расширениями и
    программами-проигрывателями; запускает веб-броузер для файлов с типом
    text/html, с неизвестным типом и при отсутствии таблицы соответствий
    """
    contenttype, encoding = mimetypes.guess_type(filename) # проверить имя
    if contenttype == None or encoding is not None:        # не определяется
        contenttype = '?/?'                                # возм. .txt.gz
    maintype, subtype = contenttype.split('/', 1)          # 'image/jpeg'
    if maintype == 'text' and subtype == 'html':
        trywebbrowser(filename, **options)                 # спец. случай
    elif maintype in mimetable:
        playknownfile(filename, mimetable[maintype], **options) # по таблице
    else:
        trywebbrowser(filename, **options)                 # другие типы

################################################################################
# программный код самопроверки
################################################################################

if __name__ == '__main__':
    # тип медиафайла известен
    playknownfile('sousa.au', audiotools, wait=True)
    playknownfile('ora-pp3e.gif', imagetools, wait=True)
    playknownfile('ora-lp4e.jpg', imagetools)

    # тип медиафайла определяется
    input('Stop players and press Enter')
    playfile('ora-lp4e.jpg')                  # image/jpeg
```

```
playfile('ora-pp3e.gif')                  # image/gif
playfile('priorcalendar.html')            # text/html
playfile('lp4e-preface-preview.html')     # text/html
playfile('lp-code-readme.txt')            # text/plain
playfile('spam.doc')                      # app
playfile('spreadsheet.xls')               # app
playfile('sousa.au', wait=True)           # audio/basic
input('Done')                             # приостановиться, если
                                          # сценарий запущен щелчком мыши
```

В наши дни медиафайлы большинства типов можно открыть с помощью веб-броузера, тем не менее этот модуль представляет собой простую основу для программ, открывающих медиафайлы с помощью более специализированных инструментов, предназначенных для обработки медиафайлов данного типа на данной платформе. Веб-броузер используется только в крайнем случае, при отсутствии других возможностей. В результате получился расширяемый сценарий проигрывания медиафайлов, специализированность и переносимость которого зависит от настроек в его таблицах соответствий.

Инструменты запуска программ, используемые этим сценарием, мы видели в предыдущих главах. Основным новшеством в этом сценарии является использование новых модулей: модуль `webbrowser` используется для открытия некоторых файлов в локальном веб-броузере, а модуль `mimetypes` применяется для определения типов медиафайлов по их именам. Поскольку они являются основой этого сценария, коротко познакомимся с ними, прежде чем запускать сценарий.

Модуль webbrowser

Модуль `webbrowser`, входящий в состав стандартной библиотеки и используемый в этом примере, предоставляет переносимый интерфейс для запуска веб-броузера из сценариев на языке Python. Он пытается отыскать подходящий веб-броузер на локальном компьютере, чтобы открыть указанный адрес URL (полное имя файла или веб-адрес). Интерфейс модуля достаточно прост:

```
>>> import webbrowser
>>> webbrowser.open_new('file://' + fullfilename) # используйте
                                                  # os.path.abspath()
```

Этот программный код откроет указанный файл в новом окне броузера, который удастся обнаружить на локальном компьютере, или возбудит исключение. Имеется возможность явно перечислить броузеры, имеющиеся в системе, и определить порядок, в каком они будут использоваться, с помощью переменной окружения `BROWSER` и функции `register`. По умолчанию модуль `webbrowser` автоматически пытается обеспечить переносимость между платформами.

Чтобы открыть файл, находящийся на локальном компьютере или на веб-сервере, строка аргумента должна иметь вид «file://...» или «http://...»

соответственно. Фактически можно передать строку URL любого вида, которую воспринимает броузер. Например, следующая инструкция откроет домашнюю страницу проекта Python в новом окне броузера:

```
>>> webbrowser.open_new('http://www.python.org')
```

Кроме всего прочего, этот модуль обеспечивает простой способ отображения документов HTML, а также медиафайлов, как демонстрируется в примере из этого раздела. К тому же этот модуль может использоваться и как самостоятельный сценарий командной строки (здесь вам поможет ключ `-m` командной строки интерпретатора, включающий путь поиска модулей), и как импортируемый инструмент:

```
C:\Users\mark\Stuff\Websites\public_html> python -m webbrowser about-pp.html
C:\Users\mark\Stuff\Websites\public_html> python -m webbrowser -n about-pp.html
C:\Users\mark\Stuff\Websites\public_html> python -m webbrowser -t about-pp.html

C:\Users\mark\Stuff\Websites\public_html> python
>>> import webbrowser
>>> webbrowser.open('about-pp.html')     # повторное использование,
True                                     # новое окно, новая вкладка
>>> webbrowser.open_new('about-pp.html') # file:// не обязательно в Windows
True
>>> webbrowser.open_new_tab('about-pp.html')
True
```

В обоих режимах применения различия между тремя формами заключаются в том, что в первом случае предпринимается попытка повторно использовать уже открытое окно броузера, если это возможно, во втором – выполняется попытка открыть новое окно, и в третьем – выполняется попытка открыть новую вкладку. Однако на практике поведение всех трех форм зависит от типа веб-броузера и в целом от типа платформы. Все три формы могут вести себя совершенно одинаково.

В Windows, например, все три просто вызывают функцию `os.startfile` по умолчанию, в результате чего создается новая вкладка в уже открытом окне Internet Explorer 8. Это объясняет, почему я не указывал полный префикс «file://» URL в предыдущем фрагменте. Формально, броузер Internet Explorer запускается, только если он зарегистрирован как инструмент открытия файлов указанного типа, в противном случае запускается специализированный инструмент для работы с такими файлами. Некоторые изображения, например, могут открываться в специализированных программах просмотра фотографий. На других платформах, таких как Unix и Mac OS X, поведение броузера отличается, и файлы, имена которых не являются полноценным адресом URL, могут не открываться, поэтому для переносимости используйте префикс «file://».

Мы еще вернемся к этому модулю далее в книге. Например, программа PyMailGUI в главе 14 будет использовать его как инструмент отображения сообщений электронной почты и вложений в формате HTML, а также отображения справки к программе. За дополнительной инфор-

мацией обращайтесь к руководству по библиотеке Python. В главах 13 и 15 мы также встретим родственную функцию `urllib.request.urlopen`, которая извлекает текст веб-страницы, находящейся по указанному адресу URL, но не открывает его в броузере – этот текст можно проанализировать, сохранить и использовать как-то иначе.

Модуль mimetypes

Чтобы сделать этот модуль проигрывателя медиафайлов еще более удобным, мы использовали в нем модуль `mimetypes`, входящий в состав стандартной библиотеки Python, который автоматически определяет тип медиафайла по его имени. Если тип может быть определен, модуль возвращает строку типа содержимого вида `type/subtype`, или `None` – если тип не определяется:

```
>>> import mimetypes
>>> mimetypes.guess_type('spam.jpg')
('image/jpeg', None)

>>> mimetypes.guess_type('TheBrightSideOfLife.mp3')
('audio/mpeg', None)

>>> mimetypes.guess_type('lifeofbrian.mpg')
('video/mpeg', None)

>>> mimetypes.guess_type('lifeofbrian.xyz')    # неизвестный тип
(None, None)
```

Выделив первую часть строки типа содержимого, можно получить обобщенный тип медиафайла, который можно использовать для выбора универсальной программы-проигрывателя. Вторая часть (подтип) сообщает, например, является текст разметкой HTML или простым текстом:

```
>>> contype, encoding = mimetypes.guess_type('spam.jpg')
>>> contype.split('/')[0]
'image'

>>> mimetypes.guess_type('spam.txt')   # подтип 'plain'
('text/plain', None)

>>> mimetypes.guess_type('spam.html')
('text/html', None)

>>> mimetypes.guess_type('spam.html')[0].split('/')[1]
'html'
```

Тут есть одна тонкость: второй элемент кортежа, возвращаемого функцией определения типа из модуля `mimetypes`, является названием кодировки, которое мы не используем здесь. Тем не менее мы не должны забывать про него – если данный элемент имеет значение, отличное от

None, это означает, что файл был сжат (с помощью gzip или compress), даже если в первом элементе кортежа возвращается тип медиафайла. Например, если файл имеет такое имя, как *spam.gif.gz*, будет считаться, что это сжатое изображение, которое не следует пытаться открывать непосредственно:

```
>>> mimetypes.guess_type('spam.gz')            # тип содержимого неизвестен
(None, 'gzip')

>>> mimetypes.guess_type('spam.gif.gz')        # не открывать напрямую!
('image/gif', 'gzip')

>>> mimetypes.guess_type('spam.zip')           # архивы
('application/zip', None)

>>> mimetypes.guess_type('spam.doc')           # файлы офисных приложений
('application/msword', None)
```

Если имя файла, передаваемое функции, содержит путь к каталогу, то эта часть имени будет игнорироваться (при определении типа используется только расширение). Этот модуль позволяет даже определять расширение имени файла по заданному типу, что можно использовать при создании имен файлов по известному типу содержимого:

```
>>> mimetypes.guess_type(r'C:\songs\sousa.au')
('audio/basic', None)

>>> mimetypes.guess_extension('audio/basic')
'.au'
```

Поэкспериментируйте с другими функциями из этого модуля, чтобы получить более полное представление о нем. Мы еще раз вернемся к модулю mimetypes в примерах FTP в главе 13, где будем определять тип передаваемых данных (текст или двоичные), а также в примерах работы с электронной почтой в главах 13, 14 и 16, где реализуем отправку, сохранение и открытие вложений.

В примере 6.23 модуль mimetypes используется для организации выбора платформозависимых команд запуска программ-проигрывателей для данного типа медиафайлов. То есть модуль выбирает проигрыватель из таблицы по указанному типу медиафайла, а затем выбирает из таблицы проигрывателей команду для данной платформы. На любом из двух этапах мы отступаем назад и запускаем веб-броузер, если не было найдено чего-то более определенного.

Использование модуля mimetypes в классе SearchVisitor

Чтобы задействовать этот модуль для управления выбором текстовых файлов в сценариях поиска, написанных нами выше в этой главе, можно просто извлекать и анализировать первый элемент кортежа, воз-

вращаемого для заданного имени файла. Например, все расширения в следующем списке считаются расширениями имен текстовых файлов (кроме расширения «.pyw», которое можно считать особым случаем):

```
>>> for ext in ['.txt', '.py', '.pyw', '.html', '.c', '.h', '.xml']:
...     print(ext, mimetypes.guess_type('spam' + ext))
...
.txt ('text/plain', None)
.py ('text/x-python', None)
.pyw (None, None)
.html ('text/html', None)
.c ('text/plain', None)
.h ('text/plain', None)
.xml ('text/xml', None)
```

Мы можем добавить этот прием в предыдущую реализацию класса `SearchVisitor`, переопределив метод `candidate` и заменив список расширений имен файлов, используемый по умолчанию, анализом возвращаемого типа модулем `mimetypes` – еще одно яркое свидетельство гибкости ООП:

```
C:\...\PP4E\Tools> python
>>> import mimetypes
>>> from visitor import SearchVisitor # или PP4E.Tools.visitor, если не .
>>>
>>> class SearchMimeVisitor(SearchVisitor):
...     def candidate(self, fname):
...         contype, encoding = mimetypes.guess_type(fname)
...         return (contype and
...                 contype.split('/')[0] == 'text' and
...                 encoding == None)
...
>>> V = SearchMimeVisitor('mimetypes', trace=0) # ключ поиска
>>> V.run(r'C:\temp\PP3E\Examples')             # корневой каталог
C:\temp\PP3E\Examples\PP3E\extras\LosAlamosAdvancedClass\day1-system\data.txt
has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py has
mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py has mimetypes
>>> V.scount, V.fcount, V.dcount
(8, 1429, 186)
```

Однако этот прием не обеспечивает абсолютную точность (в случае необходимости придется добавить логику обслуживания таких расширений, как «.pyw», которые не определяются модулем `mimetypes`) и подходит далеко не для всех клиентов (в некоторых случаях может потребоваться искать только определенные типы текстовых файлов), поэтому

данная схема не использовалась в оригинальном классе. Использование и настройку его для ваших собственных нужд мы оставим в качестве самостоятельного упражнения.

Запускаем сценарий

Если теперь запустить сценарий из примера 6.23 и все пойдет как надо, программный код самопроверки в конце сценария откроет множество аудиофайлов, изображений, текста и других файлов, находящихся в каталоге сценария, используя либо специализированные программы-проигрыватели для данной платформы, либо веб-броузер. На моем ноутбуке с Windows 7 файлы GIF и HTML открываются в новых вкладках броузера IE; файлы JPEG – в программе Windows Photo Viewer; простые текстовые файлы – в программе Notepad; файлы DOC и XLS – в Microsoft Word и Excel; а аудиофайлы – в Windows Media Player.

Однако, поскольку набор используемых программ и их поведение могут значительно отличаться для разных компьютеров, вам будет лучше самостоятельно изучить реализацию сценария и опробовать его на собственном компьютере со своими тестовыми файлами, чтобы своими глазами увидеть, что произойдет. Как обычно, тестирование можно выполнить и в интерактивной оболочке (используйте путь к пакету, как в следующем фрагменте, чтобы импортировать из другого каталога, при этом предполагается, что корневой каталог *PP4E* находится в пути поиска модулей):

```
>>> from PP4E.System.Media.playfile import playfile
>>> playfile(r'C:\movies\mov10428.mpg')              # video/mpeg
```

Мы еще будем использовать модуль `playfile` в главе 13 для открытия медиафайлов, загруженных по FTP. И снова вам может потребоваться настроить таблицы в сценарии, определив в них ассоциации со своими программами-проигрывателями. Кроме того, этот сценарий предполагает, что медиафайл находится на локальном компьютере (даже при том, что модуль `webbrowser` поддерживает удаленные файлы с именами, начинающимися с «http://») и в настоящее время не позволяет использовать различные программы-проигрыватели для большинства различных подтипов MIME (текстовые файлы являются особым случаем, где подтипы «plain» и «html» обрабатываются по-разному, но это не относится к другим типам). Этот сценарий является своего рода основой, предназначенной для дальнейшего расширения. Как обычно, изучайте и осваивайте – в конце концов, это Python.

Автоматизированный запуск программ (внешние примеры)

Наконец, кое-что для дополнительного чтения – в пакете с примерами для этой книги (доступном на сайтах, перечисленных в Предисловии)

вы найдете дополнительные системные сценарии, описать которые здесь мы не можем из-за нехватки места:

- *PP4E\Launcher.py* – содержит инструменты, используемые некоторыми программами с графическим интерфейсом, рассматриваемыми далее в этой книге, для запуска программ Python без необходимости выполнять настройку окружения. Грубо говоря, этот модуль выполняет настройку системного пути поиска файлов и пути поиска импортируемых модулей, необходимых для запуска примеров, которые наследуются порождаемыми программами. Используя этот модуль для поиска файлов и автоматической настройки окружения, пользователи могут избежать или, по крайней мере, отсрочить необходимость изучать особенности настройки окружения вручную перед запуском программ. В этом примере вы найдете не так много нового с точки зрения системных интерфейсов, однако мы еще будем ссылаться на него позднее, когда будем исследовать программы с графическим интерфейсом, использующие эти инструменты, а также родственные им инструменты запуска, о которых рассказывалось в главе 5.
- *PP4E\Launch_PyDemos.pyw* и *PP4E\Launch_PyGadgets_bar.pyw* – используют *Launcher.py* для запуска основных примеров книги без необходимости выполнять настройку окружения. Поскольку все порождаемые процессы наследуют настройки, выполненные модулем запуска, все они выполняются с соответствующими настройками путей поиска. При непосредственном запуске сценарии *PyDemos2.pyw* и *PyGadgets_bar.pyw* (которые мы будем исследовать в конце главы 10) будут использовать общесистемные настройки. Другими словами, сценарий `Launcher` эффективно скрывает особенности настройки от графических интерфейсов, заключая их в настроенную программную оболочку.
- *PP4E\LaunchBrowser.pyw* – переносимым образом отыскивает и запускает веб-броузер на локальном компьютере для просмотра содержимого локального файла или удаленной веб-страницы. Поиск броузера в предыдущей версии выполнялся с помощью инструментов из *Launcher.py*. Изначальная реализация этого модуля может быть в значительной степени заменена модулем `webbrowser` из стандартной библиотеки, который появился уже после того, как этот пример был создан (питоны думают одинаково!). В этом издании модуль `LaunchBrowser` просто анализирует аргументы командной строки для обратной совместимости и вызывает функцию `open` из модуля `webbrowser`. Примеры его использования вы найдете в справочном тексте внутри модуля или в примерах PyGadgets и PyDemos в главе 10.

На этом мы заканчиваем исследование системных инструментов. В следующей части книги мы оставим царство командной строки и перейдем к исследованию способов создания графических интерфейсов для наших программ. Позднее мы проделаем то же самое, но уже с ис-

пользованием веб-интерфейсов. Тем не менее не будем забывать, что системные инструменты, с которыми мы познакомились в этой части книги, находят применение в самых разных программах. Например, мы будем использовать потоки выполнения для выполнения длительных операций в части книги, посвященной графическим интерфейсам; потоки выполнения и процессы будут использоваться для реализации серверов в части книги, посвященной программированию для Интернета; и на протяжении всей книги будем применять файлы и системные вызовы, связанные с файлами.

Будь то сценарии командной строки, многооконные графические интерфейсы или распределенные веб-сайты, действующие по схеме клиент/сервер, системные инструменты Python всегда будут играть важную роль в вашей карьере программиста на Python.

III

Программирование графических интерфейсов

Эта часть книги демонстрирует, как применять Python для создания переносимых графических интерфейсов пользователя, в первую очередь, с помощью стандартной библиотеки Python tkinter. Следующие главы всесторонне освещают эту тему:

Глава 7

Эта глава очерчивает возможности создания графических интерфейсов, доступные в языке Python, а затем представляет учебный материал, иллюстрирующий базовые понятия программирования с использованием tkinter.

Глава 8

Эта глава начинает обзор двух частей библиотеки tkinter – набора графических элементов и сопутствующих инструментов. В первой части обзора рассказывается о наиболее простых инструментах и графических элементах библиотеки: всплывающих окнах, различных видах кнопок, изображениях и так далее.

Глава 9

Здесь продолжается обзор библиотеки, начатый в предыдущей главе. В этой главе представляется остальная часть библиотеки графических элементов tkinter, включая меню, текст, холст, полосы прокрутки, сетки, события таймера и анимацию.

Глава 10

В этой главе рассматриваются приемы программирования графических интерфейсов: здесь мы узнаем, как автоматически конструировать меню из шаблонов объектов, как запускать графический интерфейс в виде отдельной программы, как реализовать выполнение

продолжительных операций параллельно с основной программой с помощью потоков выполнения и очередей и многое другое.

Глава 11

Эта глава объединяет идеи из предыдущих глав для реализации набора интерфейсов пользователя. В ней будут представлены более крупные примеры графических интерфейсов – часов, текстовых редакторов, программ для рисования и просмотра графических изображений и других – которые также демонстрируют приемы программирования на Python в целом.

Как и в первой части книги, представленный здесь материал применим в целом ряде областей и будет снова использован в дальнейшем для создания настроенных на конкретные области применения интерфейсов в последующих главах книги. Так, примеры PyMailGUI и PyCalc в последующих главах предполагают, что вы знакомы с основами, охватываемыми здесь.

7

Графические интерфейсы пользователя

«Я здесь, я смотрю на тебя, детка»

Для большинства программных систем графический интерфейс пользователя (Graphical User Interface, GUI) стал непременной частью пакета. Даже если акроним «GUI» вам незнаком, вы, вероятно, знакомы с такими элементами, как окна, кнопки и меню, используемые при работе с программами. В действительности большая часть работы с компьютерами сегодня осуществляется с помощью того или иного графического интерфейса вида «укажи-и-щелкни». Программы, от веб-броузера до системных инструментов, стандартно оснащаются компонентами GUI, повышающими гибкость и простоту их использования.

В этой части мы узнаем, как заставлять сценарии Python порождать такие графические интерфейсы, изучая примеры программирования с помощью модуля *tkinter* – переносимой библиотеки GUI, являющейся стандартной частью системы Python и широко используемой многими программистами. Как будет показано, легкость программирования интерфейсов пользователя в сценариях Python обеспечивается как простотой языка, так и мощью его библиотек GUI. Дополнительным преимуществом графических интерфейсов, запрограммированных на Python с использованием tkinter, является то, что они автоматически переносимы на большинство компьютерных систем.

Темы программирования GUI

Поскольку графические интерфейсы представляют собой большую область, я хочу сказать еще несколько слов об этой части книги. Чтобы сделать усвоение проще, темы программирования GUI разделены между следующими пятью главами этой книги:

- Данная глава начинается с краткого учебника по библиотеке tkinter, знакомящего с основами ее использования. 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 освещаются все основы: обработка событий, менеджер компоновки `pack`, использование наследования и композиции в GUI и многое другое. Как будет показано, объектно-ориентированное программирование (ООП) не является обязательным для tkinter, но оно делает графические интерфейсы структурированными и многократно используемыми.
- Главы 8 и 9 представляют обзор набора графических элементов (виджетов) в библиотеке tkinter.[1] В общих чертах, в главе 8 представлены простые графические элементы, а в главе 9 – более сложные графические элементы и связанные с ними инструменты. Здесь рассматривается большинство привычных элементов интерфейсов: ползунки, меню, диалоги, изображения и тому подобное. Эти две главы не могут служить полным справочником по библиотеке tkinter (который вполне мог бы превратиться в весьма объемную книгу), но их должно хватить, чтобы начать создавать серьезные графические интерфейсы на языке Python. Примеры в этих главах сосредоточены на графических элементах и инструментах tkinter, но попутно исследуется поддержка повторного использования программного кода в Python.
- В главе 10 представлены более сложные приемы программирования GUI. Здесь будут исследованы приемы автоматизации типичных задач, решаемых в графических интерфейсах на языке Python. И хотя библиотека tkinter является полнофункциональной, тем не менее небольшое количество многократно используемого программного кода на языке Python сможет сделать интерфейсы еще более мощными и простыми в использовании.
- Глава 11 завершает эту часть книги представлением нескольких программ с графическим интерфейсом, в которых используются приемы программирования и графические элементы, показанные в четырех предшествующих главах. Здесь мы узнаем, как реализовать текстовые редакторы, средства просмотра графических изображений, часы и многое другое.

---

[1] Термин «набор графических элементов» означает объекты, используемые для создания привычных элементов интерфейса «укажи-и-щелкни» – кнопок, ползунков, полей ввода и так далее. В библиотеке tkinter определяются классы Python, соответствующие всем графическим элементам, к которым вы привыкли в графических интерфейсах. Помимо графических элементов в tkinter присутствуют также инструменты для решения иных задач, таких как планирование событий, ожидание появления данных в сокетах и так далее.

Поскольку графические интерфейсы являются инструментами, используемыми в самых разных областях, другие примеры их реализации будут появляться на протяжении всей оставшейся части книги. Например, позднее мы увидим законченные графические интерфейсы для работы с электронной почтой и калькуляторы, а также простой клиент FTP с графическим интерфейсом. Дополнительные примеры, такие как средства представления деревьев и броузеры таблиц, доступны в виде внешних примеров, входящих в состав пакета с примерами. В главе 11 приводится список ссылок на другие примеры использования tkinter в данной книге.

Закончив исследование графических интерфейсов, мы в четвертой части узнаем, как конструировать пользовательские интерфейсы внутри веб-броузера, используя HTML и сценарии на языке Python, выполняющиеся на веб-сервере, – совершенно иная модель со своими достоинствами и недостатками, понимать которые просто необходимо. Далее в этой главе описываются такие новейшие технологии, как RIA (Rich Internet Application – полнофункциональные интернет-приложения), основанные на модели веб-броузера и способные предложить еще более широкие возможности в конструировании интерфейсов.

А пока сосредоточимся на более традиционных графических интерфейсах, известных как «настольные» приложения или как «автономные» графические интерфейсы. Как мы увидим далее, в части книги, посвященной созданию сценариев для Интернета, когда встретимся с графическими интерфейсами клиентов FTP и электронной почты, такие программы часто устанавливают сетевые соединения, необходимые для их работы.

Запуск примеров

Сразу же хочу отметить одно обстоятельство: в большинстве своем графические интерфейсы являются динамическими и интерактивными, а лучшее, что я могу здесь сделать, – это показать статические снимки экрана, представляющие избранные состояния взаимодействий, реализуемых такими программами. Для большинства примеров это несправедливо. Если вы еще не работаете с примерами, настоятельно рекомендую запускать примеры графических интерфейсов в этой и последующей главах самостоятельно.

В Windows стандартная установка Python имеет встроенную поддержку tkinter, поэтому все эти примеры должны работать сразу. Mac OS X также поставляется со встроенной поддержкой tkinter в Python. В других системах Python с поддержкой tkinter либо уже входит в состав системы, либо доступен для установки (подробности смотрите в файле верхнего уровня *README-PP4E.txt* в дереве примеров для этой книги). Несмотря на то, что вам, возможно, потребуется установить дополнительные пакеты, чтобы обеспечить работоспособность tkinter, это стоит сделать, потому что возможность поэкспериментировать с этим про-

граммами является отличным способом изучения не только программирования графических интерфейсов, но и самого языка Python.

Смотрите также замечания о переносимости примеров к этой книге в Предисловии. Несмотря на то, что и сам язык Python, и библиотека tkinter в значительной степени являются нейтральными к платформе, на которой они используются, вам, возможно, придется столкнуться с некоторыми небольшими проблемами, если вы будете пытаться запускать примеры на платформах, которые не использовались в процессе работы над этой книгой. В Mac OS X, например, могут обнаружиться тонкие отличия в работе некоторых примеров. Обязательно загляните на веб-сайт книги, где можно найти дополнительные указания и исправления, которые помогут использовать примеры на других платформах.

Кто-нибудь заметил, что «GUI» совпадает с тремя первыми буквами в имени «GUIDO»?

Создатель Python, Гвидо ван Россум (Guido van Rossum), изначально не ставил своей целью создание инструмента для разработки GUI, но из-за простоты использования Python и краткости цикла разработки он выдвинулся на одну из первых ролей. С точки зрения реализации, графические интерфейсы в Python опираются на использование расширений на языке 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. Это может рассматриваться как пример взаимодействия Python с внешними библиотеками.

С практической точки зрения графические интерфейсы являются жизненно важной частью современных систем и идеальной областью применения таких инструментов, как Python. Как будет показано далее, простота синтаксиса Python и его объектно-ориентированные возможности 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 использовании обычных технологий разработки. На практике обычно можно за считаные секунды сделать изменения в GUI, созданном на Python, и посмотреть на их результат. Не пытайтесь добиться такой же скорости на C или C++ !

Различные возможности создания GUI в Python

Прежде чем погрузиться в tkinter, рассмотрим перспективные варианты разработки GUI в Python в целом. Поскольку Python показал себя вполне подходящим инструментом для работы с графическим интерфейсом, в последние годы в этой области наблюдалась высокая активность. На практике, хотя в качестве инструмента для создания GUI в Python чаще всего используется библиотека tkinter, на сегодняшний день существуют и другие способы программирования пользовательских интерфейсов в Python. Некоторые являются специфическими для Windows или X Window,[1] другие представляют собой решения для нескольких платформ, и у всех них есть свои приверженцы и свои сильные стороны. Чтобы быть справедливыми ко всем вариантам, приведем краткий перечень инструментов создания графических интерфейсов, доступных программистам Python на момент написания этих строк:

tkinter

Библиотека для разработки графических интерфейсов, распространяемая с открытыми исходными текстами, ставшая де-факто стандартом для разработки переносимых графических интерфейсов на языке Python. Сценарии на языке Python, использующие tkinter для построения GUI, выполняются переносимым образом в Windows, X Window (Unix и Linux) и Macintosh OS X, создавая на каждой из этих платформ графический интерфейс, со свойственным им внешним видом. Кроме того, она может легко расширяться программным кодом на языке Python, а также имеет множество дополнительных пакетов, таких как *Pmw* (сторонняя библиотека виджетов); *Tix* (еще одна библиотек виджетов, ныне ставшая стандартной частью Python); *PIL* (расширение для обработки изображений) и *ttk* (библиотека виджетов Tk, поддерживающих темы оформления, также ставшая стандартной частью Python, начиная с версии 3.1). Подробнее о подобных расширениях рассказывается ниже в этом введении.

Библиотека Tk, на которой основана библиотека tkinter, является стандартом в мире открытого программного обеспечения в це-

[1] В данной книге «Windows» относится к интерфейсу Microsoft Windows, распространенному на PC, а «X Window» относится к интерфейсу X11, чаще всего встречающемуся на платформах Unix и Linux. Эти два интерфейса в целом привязаны к платформам Microsoft и Unix (и Unix-подобным), соответственно. Существует возможность выполнения X Window поверх операционной системы Microsoft и эмуляции Windows в Unix и Linux, но эта возможность используется достаточно редко. Чтобы добавить туману, замечу, что Mac OS X обеспечивает поддержку tkinter в X Window и в родной графической подсистеме Aqua, в дополнение к платформозависимой поддержке сосоа (хотя обычно особенности OS X не слишком выделяются в коктейле Unix-подобных систем).

лом и используется также языками сценариев Perl, Ruby, PHP, Common Lisp и Tcl, благодаря чему количество пользователей этой библиотеки может исчисляться миллионами. Промежуточная библиотека, связывающая Python и Tk, дополняет последнюю простой объектной моделью Python – благодаря ей виджеты Tk из строковых команд превратились в легко настраиваемые объекты. Библиотека tkinter в Python 3.X приняла форму пакета с вложенными модулями, обеспечивающими группировку некоторых из ее инструментов по функциональности (ранее, в Python 2.X, эта библиотека имела форму модуля *tkinter*, но была переименована в соответствии с общепринятыми соглашениями об именовании и реструктурирована для обеспечения иерархической организации).

tkinter – зрелая, надежная, широко используемая и хорошо документированная библиотека. Она включает около 25 основных типов виджетов, а также различные диалоги и другие инструменты. Издана книга, посвященная описанию библиотеки, а кроме того, имеется огромное количество опубликованной документации с описанием tkinter и Tk. Также важно, что так как tkinter основана на библиотеке, разработанной для использования в языках сценариев, она является относительно легковесным набором инструментов и поэтому прекрасно сочетается с языком Python.

Благодаря таким характеристикам библиотека tkinter распространяется в составе Python как модуль стандартной библиотеки и стала основой стандартной интегрированной среды разработки IDLE с графическим интерфейсом. Фактически библиотека tkinter является единственным набором инструментов для создания графических интерфейсов, ставшим частью Python, – все остальные в этом списке являются сторонними расширениями. На некоторых платформах (включая Windows, Mac OS X и большинство Linux и Unix-подобных систем) библиотека Tk также распространяется в составе Python. Вы можете с достаточной степенью уверенности рссчитывать, что к моменту запуска сценария библиотека tkinter будет присутствовать на компьютере, а при необходимости вы сможете обеспечить это, скомпилировав свой графический интерфейс в автономный выполняемый файл с помощью таких инструментов, как PyInstaller и py2exe (подробности ищите в Интернете).

Несмотря на простоту библиотеки tkinter, ее текстовые и графические виджеты обеспечивают достаточно широкие возможности для реализации веб-страниц, трехмерных изображений и анимации. Кроме того, на сегодняшний день многие системы предоставляют построители графических интерфейсов для связки Python/tkinter, включая GUI Builder (ранее входивший в состав среды разработки Komodo IDE и родственной ей SpecTCL), Rapyd-Tk, xRope и другие (этот перечень имеет свойство значительно изменяться с течением времени; более полный перечень вы найдете на странице *http://wiki.python.org/moin/GuiProgramming* или поискав самостоятельно в Ин-

тернете). Однако, как будет показано далее, библиотека tkinter во многих случаях оказывается настолько удобна в использовании, что построители графических интерфейсов не получили широкого распространения. В справедливости этого утверждения мы убедимся, как только покинем область статических интерфейсов, которые обычно поддерживаются построителями.

wxPython

Интерфейс Python к библиотеке wxWidgets с открытыми исходными текстами (ранее называвшейся wxWindows), которая представляет собой переносимую структуру классов GUI, первоначально созданную для использования из программ на языке C++. Система wxPython является модулем расширения, служащим оболочкой для классов wxWidgets. По общему мнению эта библиотека превосходно подходит для создания сложных интерфейсов и сегодня является, вероятно, вторым по популярности инструментом создания графических интерфейсов в Python после tkinter. Графические интерфейсы, созданные с применением wxPython, переносимы на Windows и Unix-подобные платформы и Mac OS X.

Поскольку интерфейс wxPython опирается на библиотеку классов C++, многие полагают, что он более сложен в использовании, чем библиотека tkinter: он предоставляет доступ к сотням классов, для чего требуется прибегать к объектно-ориентированному стилю программирования, и имеет архитектуру, которая некоторым напоминает библиотеку классов MFC в Windows. Применение wxPython часто требует от программистов писать больше программного кода, отчасти потому, что этот интерфейс обладает более широкими функциональными возможностями, а отчасти потому, что именно такой образ мышления он унаследовал от библиотеки C++, лежащей в его основе.

Кроме того, часть документации wxPython ориентирована на программистов, использующих язык C++. Впрочем, недавно эта ситуация немного улучшилась – после выхода книги, посвященной wxPython. Библиотеке tkinter, напротив, посвящена книга, огромные разделы в других книгах о языке Python и еще масса литературы по библиотеке Tk, лежащей в ее основе. Однако, поскольку мир книг о языке Python в последние годы расширялся достаточно динамично, вам следует тщательно подходить к выбору литературы – некоторые книги со временем устаревают, но регулярно появляются новые.

В обмен на повышенную сложность библиотека wxPython обеспечивает набор мощных инструментов. В состав wxPython входит более богатый набор виджетов, чем в библиотеку tkinter, включая деревья и компоненты просмотра HTML – чтобы получить такие же компоненты при использовании tkinter, может потребоваться задействовать такие расширения, как Pmw, Tix или ttk. Кроме того, некоторым нравится, как выглядят графические интерфейсы, созданные

с помощью wxPython. BoaConstructor и wxDesigner среди других возможностей предоставляют построители графических интерфейсов, которые генерируют программный код для wxPython. Некоторые инструменты в библиотеке wxWidgets также поддерживают операции, не имеющие отношения к графическим интерфейсам. Чтобы быстро посмотреть, как выглядят виджеты wxPython и соответствующий программный код, запустите демонстрационный пример, который поставляется вместе с wxPython (смотрите страницу *http://wxpython.org/* или поищите в Интернете самостоятельно).

PyQt

Интерфейс Python к библиотеке Qt (ныне принадлежит Nokia, ранее принадлежала компании Trolltech), занимающей, пожалуй, третье место среди наиболее часто используемых инструментов GUI для Python. PyQt – это полноценная библиотека создания графических интерфейсов, которая на сегодняшний день переносима в Windows, Mac OS X, Unix и Linux. Подобно wxPython, библиотека PyQt в целом более сложна в использовании, чем tkinter, и при этом обладает более богатыми возможностями – она содержит сотни классов и тысячи функций и методов. Библиотека Qt зародилась и выросла в Linux, но со временем была перенесена и на другие системы. Вследствие своего происхождения расширения PyQt и PyKDE предоставляют доступ к библиотекам KDE (PyKDE требует библиотеку PyQt). Системы BlackAdder и Qt Designer предоставляют построители GUI для PyQt.

Самым важным, пожалуй, недостатком Qt в прошлом считалась неполная открытость библиотеки для коммерческого использования. Сегодня библиотека Qt распространяется не только под коммерческой лицензией, но и под открытыми лицензиями GPL и LGPL. Версии, распространяемые под LGPL и GPL, являются открытыми и следуют требованиям лицензии GPL (GPL накладывает требования, отсутствующие в лицензии BSD, под которой распространяется сам Python, – согласно GPL, например, вы должны сделать исходные тексты программ доступными для конечных пользователей).

PyGTK

Интерфейс Python к GTK, переносимой библиотеке GUI, первоначально использовавшейся как ядро оконной системы Gnome в Linux. Пакеты расширений gnome-python и PyGTK экспортируют функции в инструментальных наборах Gnome и GTK для использования в сценариях Python. К моменту написания этих строк интерфейс PyGTK поддерживал возможность работы в Windows и в POSIX-совместимых системах, таких как Linux и Mac OS X (согласно документации, в настоящее время требуется, чтобы в Mac OS был установлен X-сервер, при этом разрабатывается версия для Mac).

Jython

Jython (система, известная ранее, как JPython) является реализацией Python для Java, которая компилирует исходный программный код Python в байт-код Java и обеспечивает сценариям Python беспрепятственный доступ к библиотекам классов Java на локальном компьютере. Благодаря этому библиотеки для построения графических интерфейсов на языке Java, такие как `swing` и `awt`, дают еще один способ построения GUI на языке Python, выполняемом в системе JPython. Очевидно, такие решения являются специфическими для Java, и их переносимость ограничена переносимостью языка Java и его библиотек. Кроме того, следует отметить, что `swing` является самым крупным и самым сложным способом создания GUI в Python. Кроме того, существует новый пакет под названием `jtkinter`, который является версией tkinter для Jython, использующей Java JNI, – если он установлен, сценарии на языке Python смогут также использовать tkinter для построения GUI в Jython. Еще раз с Jython мы встретимся в главе 12, когда будем знакомиться с его ролью в Интернете.

IronPython

Очень напоминающая Jython, система IronPython является реализацией языка Python для окружения .NET, которая, кроме всего прочего, компилирует программы на языке Python в байт-код .NET, что также позволяет сценариям Python использовать возможности конструирования графических интерфейсов, имеющиеся в .NET Framework. Вы пишете программный код на языке Python, но для конструирования интерфейсов и приложений в целом используете компоненты C#/.NET. Программный код на IronPython может выполняться в Windows, под управлением .NET, и в Linux, под управлением Mono, реализации .NET, и Silverlight, клиентской платформы полнофункциональных интернет-приложений (RIA) для веб-броузеров (обсуждается далее).

PythonCard

Построитель и библиотека GUI с открытыми исходными текстами, реализованные поверх wxPython. Считается одним из самых близких Python-эквивалентов того вида построителей GUI, которые хорошо знакомы разработчикам на Visual Basic. PythonCard позиционируется разработчиками, как конструктор GUI для создания на языке Python кроссплатформенных приложений, способных выполняться в Windows, Mac OS X и Linux.

Dabo

Построитель графических интерфейсов с открытыми исходными текстами, также реализованный на основе wxPython, но не только. Dabo – это переносимый, трехуровневый, кроссплатформенный фреймворк

для разработки настольных приложений на языке Python, созданный по образу и подобию Visual FoxPro. Трехуровневая организация обеспечивает возможность доступа к базе данных, реализации бизнес-логики и пользовательского интерфейса. Открытая архитектура способна поддерживать различные типы баз данных и механизмов создания графических интерфейсов (wxPython, tkinter и даже HTML через HTTP).

Полнофункциональные интернет-приложения (Rich Internet Applications, RIA)

Хотя веб-страницы, отображаемые с помощью HTML, также могут считаться своего рода пользовательским интерфейсом, они исторически обладают массой ограничений, что затрудняет их отнесение к категории универсальных графических интерфейсов. Тем не менее некоторые специалисты склонны расширять рамки этой категории и включать в нее системы, обеспечивающие интерфейсам на основе броузеров более высокую динамичность по сравнению с традиционными веб-страницами. Поскольку такие системы предоставляют комплекты виджетов, отображаемых веб-броузерами, они способны предложить те же преимущества переносимости, которые свойственны веб-страницам в целом.

Это новое поколение инструментов называется модным термином *полнофункциональные интернет-приложения* (Rich Internet Applications, RIA). В их число входят AJAX и фреймворки, ориентированные на широкое применение JavaScript на стороне клиента, такие как:

Flex

Фреймворк с открытыми исходными текстами от компании Adobe и часть платформы Flash.

Silverlight

Фреймворк от корпорации Microsoft, который также реализован в Linux в виде фреймворка Moonlight для Mono, – доступный из программного кода на языке Python при использовании описанной выше системы IronPython.

JavaFX

Платформа Java для построения RIA, которые способны выполняться на различных связанных между собой устройствах.

pyjamas

Версия фреймворка Google Web Toolkit, опирающегося на использование AJAX, реализованная на языке Python, в состав которой входит набор виджетов пользовательского интерфейса и компилятор с языка Python на язык JavaScript, что позволяет выполнять сценарии в броузере, на стороне клиента.

Свои решения в этой области предлагает и стандарт *HTML5*, находящийся в разработке.

Вообще говоря, веб-броузеры сами являются «настольными» приложениями с графическим интерфейсом, но они получили гораздо более широкое распространение, чем библиотеки GUI, и благодаря инструментам RIA способны отображать другие графические интерфейсы. С помощью таких фреймворков можно строить графические интерфейсы, основанные на использовании виджетов, но за ними стоят накладные расходы, связанные с необходимостью сетевых взаимодействий, и они часто подразумевают использование более тяжеловесного программного комплекса, чем традиционные инструменты создания графических интерфейсов. В действительности, чтобы превратить броузеры в универсальные платформы отображения графических интерфейсов, полнофункциональные интернет-приложения предполагают использование дополнительных программных уровней и зависимостей и даже нескольких языков программирования. Поэтому, а также потому, что далеко не все занимаются разработкой веб-приложений (независимо от того, что вы могли об этом слышать), мы не будем включать их в рассмотрение традиционных автономных/настольных графических интерфейсов в этой части книги.

Дополнительные сведения о полнофункциональных интернет-приложениях и о пользовательских интерфейсах на основе веб-броузеров вы найдете в части книги, посвященной разработке сценариев для Интернета. К тому же обязательно следите за новостями и веяньями по этой теме. Интерактивность, которую обеспечивают эти инструменты, является ключевой составляющей того, что иногда называют «Web 2.0», делая больший акцент на Web, а не на традиционные графические интерфейсы. Однако, так как здесь для нас больший интерес представляют последние (и поскольку взаимодействие с пользователем – это взаимодействие с пользователем, независимо от используемого профессионального жаргона), мы отложим дальнейшее обсуждение этой темы до следующей части книги.

Платформозависимые инструменты

Помимо переносимых инструментов, таких как tkinter, wxPython и PyQt, и платформонезависимых решений, таких как полнофункциональные интернет-приложения, большинство основных платформ обладают также непереносимыми средствами создания графических интерфейсов на языке Python. Например, в Macintosh OS X имеется интерфейс PyObjC для Python, обеспечивающий доступ к фреймворку Objective-C/Cocoa компании Apple, который составляет основу разработки многих приложений для Mac. В Windows имеется расширение PyWin32 для Python, включающее обертки для доступа к фреймворку Microsoft Foundation Classes (MFC) (библиотека,

включающая интерфейсные компоненты), а также Pythonwin – пример типичной программы на основе MFC, реализующей среду разработки на языке Python с графическим интерфейсом. Несмотря на то, что технически в Linux имеется поддержка .NET, тем не менее система IronPython, упоминавшаяся выше, предлагает дополнительные возможности, которые могут использоваться только в Windows.

За дополнительными подробностями обо всех перечисленных инструментах обращайтесь на веб-сайты этих проектов. Существуют и другие, менее известные инструментарии GUI для Python, а к тому времени, когда вы будете читать эту книгу, наверняка появятся новые (например, IronPython стал новинкой для третьего издания, а для четвертого издания новинкой стали инструменты реализации полнофункциональных интернет-приложений). Кроме того, перечень доступных пакетов постоянно изменяется. Свежий список имеющихся инструментов можно найти с помощью поисковых систем, а также на сайте *http://www.python.org* и в каталоге PyPI пакетов сторонних разработчиков, поддерживаемом там же.

Обзор tkinter

Однако все эти способы создания графических интерфейсов сегодня значительно опережает библиотека tkinter как стандарт де-факто реализации переносимых пользовательских интерфейсов на Python, которой и посвящена эта часть книги. Доводы в пользу использования этого подхода уже приводились в главе 1 – я предпочел подробно описать один инструментарий вместо поверхностного обзора нескольких. Кроме того, большинство концепций программирования с использованием библиотеки tkinter, с которыми вы познакомитесь здесь, можно с легкостью перенести на любые другие инструменты GUI.

Практические преимущества tkinter

Но как бы то ни было, существуют более прагматичные причины, объясняющие, почему библиотека tkinter де-факто стала в мире Python стандартом разработки переносимых графических интерфейсов. Такие преимущества, как доступность, переносимость, простота получения, документированность и наличие расширений, делают ее наиболее широко используемым решением для Python в области GUI в течение многих лет:

Доступность

Библиотека tkinter в целом рассматривается как *облегченный набор инструментов* и одно из наиболее простых решений в области разработки графических интерфейсов для Python из имеющихся на сегодняшний день. В отличие от более крупных фреймворков, библиотека tkinter позволяет сразу же приступить к работе с ней, без

необходимости предварительно осваивать значительно более крупные модели взаимодействия классов. Как будет показано далее, программист может создать с помощью tkinter простой интерфейс, написав всего несколько строк программного кода, и постепенно наращивать его возможности до достижения промышленного уровня. Несмотря на всю простоту прикладного интерфейса библиотеки tkinter, она позволяет добавлять новые виджеты, написанные на языке Python, или подключать дополнительные расширения, такие как Pmw, Tix и ttk.

Переносимость

Сценарий на языке Python, в котором графический интерфейс строится с помощью библиотеки tkinter, будет работать без изменений на всех основных современных оконных платформах: Microsoft Windows, X Window (в Unix и Linux) и Macintosh OS X (а также в классической версии Mac). Более того, этот сценарий создаст интерфейс, внешний вид которого будет привычен пользователям каждой из этих платформ. Эта особенность развивалась по мере того как библиотека Tk становилась все более зрелой. Графический интерфейс, реализованный сценарием Python/tkinter, в Windows выглядит, как должен выглядеть интерфейс программы для Windows; в Unix и Linux обеспечивает такое же взаимодействие с пользователем, но демонстрирует внешний вид, знакомый пользователям X Window; и на Mac он выглядит так, как должна выглядеть программа Mac.

Простота получения

tkinter является модулем стандартной библиотеки Python, поставляемой вместе с интерпретатором. Если Python установлен на вашем компьютере, то у вас есть доступ и к библиотеке tkinter. Более того, в большинство пакетов установки Python (включая стандартный пакет установки Python для Windows, пакет установки для Mac и пакеты установки для большинства дистрибутивов Linux) уже включена поддержка tkinter. Благодаря этому сценарии, написанные с использованием модуля tkinter, сразу могут работать с большинством интерпретаторов Python, не требуя дополнительных действий по установке. Библиотека tkinter также в целом лучше поддерживается, чем существующие сегодня альтернативные пакеты. Поскольку задействованная в ней библиотека Tk используется также языками программирования Tcl и Perl (и многими другими), ей уделяется больше внимания и усилий разработчиков, чем другим имеющимся инструментариям.

Естественно, при использовании инструментов разработки графических интерфейсов важную роль играют и другие факторы, такие как наличие документации и расширений. Рассмотрим вкратце, что можно в этом отношении сказать о библиотеке tkinter.

Документация tkinter

В данной книге исследуются основы использования tkinter и большинство виджетов, чего должно быть достаточно, чтобы приступить к созданию графических интерфейсов на языке Python. С другой стороны, книга не является исчерпывающим справочником по библиотеке tkinter или по расширениям к ней. К счастью, когда я пишу этот абзац, в продаже есть по крайней мере одна книга, посвященная использованию tkinter в Python, и готовятся к выходу другие (подробности ищите в Интернете). Кроме книг можно найти электронную документацию по tkinter – полный комплект руководств по tkinter в настоящее время присутствует на сайте *http://www.pythonware.com/library*.

Кроме того, поскольку инструментарий Tk, используемый в tkinter, также де-факто является стандартом в сообществе открытого программного обеспечения в целом, можно использовать и другие источники документации. Например, библиотека Tk принята также к использованию в языках программирования Tcl и Perl, поэтому книги и документация по Tk, написанные для этих двух языков, могут также непосредственно использоваться при использовании связки Python/tkinter (правда, при этом необходимо будет учитывать синтаксические различия).

Честно говоря, я изучал библиотеку tkinter по книгам и справочникам Tcl/Tk – просто замените строки Tcl объектами Python, и вы получите в свое распоряжение дополнительные библиотеки справочников (чтение документации по Tk облегчит руководство по преобразованию Tk в tkinter, представленное в виде табл. 7.2 в конце данной главы). Например, книга «Tcl/Tk Pocket Reference» («Карманный справочник по Tcl/Tk»), выпущенная издательством 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.

После того как вы изучите основы, вы также сможете немало почерпнуть из примеров. В Интернете можно найти множество примеров демонстрационных программ, использующих tkinter, помимо тех, что будут представлены в этой книге. Даже в составе дистрибутива с исходными текстами Python имеется несколько демонстрационных программ, в подкаталоге *Demos\tkinter*. Среда разработки IDLE с графическим интерфейсом, упоминающаяся в следующем разделе, также представляет интерес для изучения.

Расширения для tkinter

Благодаря широкому использованию библиотеки tkinter, программистам доступны готовые расширения, предназначенные для работы с ней или дополняющие ее. На момент написания этих строк некоторые из них еще не были доступны для версии Python 3.X. Например:

Pmw

Python Mega Widgets является расширением, предназначенным для создания составных виджетов высокого уровня в Python с помощью модуля tkinter. Он расширяет прикладной интерфейс tkinter набором более сложных графических элементов для разработки улучшенных графических интерфейсов, и предоставляет основу для создания собственных виджетов. В число готовых и расширяемых графических элементов, имеющихся в пакете, входят блокноты, комбинированные списки, элементы выбора, панели, окна с прокруткой, диалоговые окна, виджеты группировки кнопок, всплывающие подсказки и интерфейс к пакету Blt для построения графиков.

Прикладной интерфейс графических элементов («мегавиджетов») Pmw напоминает интерфейс базовых графических элементов tkinter, поэтому в сценариях на языке Python могут совместно использоваться элементы Pmw и стандартные элементы tkinter. Кроме того, расширение Pmw написано исключительно на языке Python и потому не требует наличия компилятора с языка C или установки дополнительных инструментов. Чтобы ознакомиться с виджетами из этого расширения, а также посмотреть на программный код их реализации, запустите сценарий *demos\All.py*, входящий в дистрибутив Pmw. Получить Pmw можно по адресу: *http://pmw.sourceforge.net*.

Tix

Tix – это коллекция из более чем 40 улучшенных виджетов, изначально написанных для Tcl/Tk, но теперь доступных для использования в программах Python/tkinter. В настоящее время этот пакет входит в состав стандартной библиотеки Python под названием `tkinter.tix`. Подобно Tk, библиотека Tix, используемая модулем, распространяется в составе дистрибутива Python для Windows. Другими словами, во время установки Python в Windows одновременно устанавливается и библиотека Tix с дополнительными виджетами.

Пакет Tix содержит множество тех же графических элементов, которые входят в состав Pmw, включая счетчики, деревья, блокноты с вкладками, всплывающие подсказки, панели и многие другие. За дополнительной информацией обращайтесь к разделу о пакете Tix в руководстве по библиотеке Python. Чтобы быстро посмотреть, как выглядят виджеты из этого пакета, а также взглянуть, как они используются в программном коде, воспользуйтесь демонстрационной программой *tixwidgets.py* в каталоге *Demo\tix*, в дистрибутиве с исходными текстами Python (этот каталог не устанавливается по умолчанию в Windows и часто изменяется, но его можно отыскать после распаковки дистрибутива с исходными текстами, полученного с сайта проекта Python.org).

ttk

Библиотека ttk виджетов Tk, поддерживающих темы оформления, – это относительно новый набор виджетов, в которых предпринята попытка отделить реализацию поведения от реализации их внешнего вида. Классы виджетов сохраняют информацию о своем состоянии и поддерживают систему обратных вызовов, при этом внешний вид элементов реализуется отдельно, с применением тем оформления. Подобно Tix, это расширение начинало свое существование как отдельная библиотека, но совсем недавно, в версии Python 3.1, было интегрировано в стандартную библиотеку Python как модуль `tkinter.ttk`.

Кроме того, подобно Tix, это расширение включает улучшенные виджеты, часть из которых отсутствует в стандартной библиотеке tkinter. Точнее, ttk содержит 17 виджетов, из которых 11 уже присутствуют в библиотеке и предназначены для замены некоторых стандартных виджетов, а 6 – являются новыми: `Combobox`, `Notebook`, `Progressbar`, `Separator`, `Sizegrip` и `Treeview`. В двух словах: чтобы задействовать новые виджеты ttk, сценарии должны импортировать их из модуля ttk после импортирования модуля tkinter и вместо настройки самих виджетов настроить объекты стилей, которые могут использоваться совместно несколькими виджетами.

Как будет показано далее в этой главе, имеется возможность обеспечить единство оформления с виджетами из стандартной библиотеки tkinter – за счет создания подклассов виджетов с применением обычных приемов ООП (смотрите раздел «Настройка виджетов с помощью классов» ниже). При этом ttk предлагает дополнительные возможности оформления и улучшенные типы виджетов. За дополнительной информацией о виджетах ttk обращайтесь к соответствующему разделу в руководстве по библиотеке Python или поищите в Интернете – эта книга описывает основы использования tkinter, а расширения tix и ttk слишком большие, чтобы их можно было охватить с достаточной степенью подробности.

PIL

Python Imaging Library (PIL) является расширением с открытыми исходными текстами, добавляющим в Python дополнительные средства для работы с графикой. Помимо всего прочего, он добавляет инструменты для создания миниатюр изображений, их трансформации и преобразования, расширяет базовый набор графических объектов tkinter и обеспечивает поддержку отображения многих типов графических файлов. Расширение PIL, например, позволяет графическим интерфейсам на базе tkinter отображать изображения в форматах JPEG, TIFF и PNG, не поддерживаемых базовыми инструментами tkinter (без дополнительных расширений, библиотека tkinter поддерживает только формат GIF и некоторые растровые форматы). Более подробное описание и примеры использования это-

го расширения вы найдете в конце главы 8 – мы будем использовать это расширение в некоторых примерах, связанных с обработкой изображений. Расширение PIL можно найти на странице проекта *http://www.pythonware.com* или выполнив поиск в Интернете.

IDLE

Интегрированная среда разработки на языке Python IDLE сама написана на Python и tkinter. Она поставляется и устанавливается вместе с пакетом Python (если у вас свежий интерпретатор Python, то должна быть и среда IDLE, – в Windows щелкните на кнопке Пуск (Start), выберите пункт меню Все программы (All Programs), щелкните на элементе меню Python и вы увидите ее). Как среда разработки IDLE предоставляет текстовые редакторы с подсветкой синтаксиса, интерфейс отладки «указал-и-щелкнул» и многое другое. Эта среда может служить примером использования tkinter.

Другие

Многие расширения, предоставляющие инструменты визуализации для Python, основаны на библиотеке tkinter и ее виджете холста. Дополнительные примеры расширений библиотеки tkinter можно найти на веб-сайте PyPI или воспользовавшись поисковыми системами.

Если вы собираетесь заниматься коммерческой разработкой графических интерфейсов на основе tkinter, вам, вероятно, следует ознакомиться с такими расширениями, как Pmw, PIL, Tix и ttk, после изучения основ tkinter в данной книге. Они помогут сэкономить время разработки и добавить блеска в ваши интерфейсы. Самые последние новости и ссылки смотрите на сайтах, посвященных Python, указанных выше.

Структура tkinter

С технической точки зрения, библиотека tkinter является интегрирующей системой, требующей некоторой особой организации программ. Что это означает, мы увидим чуть ниже, а пока кратко познакомимся с некоторыми терминами и понятиями, лежащими в основе программирования графических интерфейсов на языке Python.

Структура реализации

Строго говоря, tkinter является просто названием интерфейса к Tk – библиотеке GUI, первоначально написанной для использования с языком программирования Tcl и разработанной создателем Tcl Джоном Оустерхаутом (John Ousterhout). Модуль tkinter обращается к библиотеке Tk, которая в свою очередь обращается к оконной системе: Microsoft Windows, X Window в Unix или к графической подсистеме Macintosh. Переносимость библиотеки tkinter фактически зависит от переносимости библиотеки Tk.

tkinter – программный слой поверх Tk, позволяющий сценариям на языке Python обращаться к библиотеке Tk, конструирующей и настра-

ивающей интерфейсы и возвращающей управление обратно в сценарии Python, которые обрабатывают события, генерируемые пользователем (например, щелчки мышью). Таким образом, обращения к графическому интерфейсу из сценария Python направляются в tkinter, а затем в Tk; события, возникающие в графическом интерфейсе, направляются из Tk в tkinter, а затем обратно в сценарий Python. В главе 20 мы встретимся с этими переходами под именами, которые они имеют в интеграции с языком C: *расширение* и *встраивание*.

Технически в настоящее время библиотека tkinter организована как комбинация файлов пакета `tkinter` на языке Python и модуля расширения с именем `_tkinter`, который написан на языке C. Модуль `_tkinter` обращается к библиотеке Tk, используя инструменты расширений, и производит обратные вызовы объектов Python, используя инструменты встраивания, – модуль `tkinter` просто добавляет объектно-ориентированный интерфейс поверх `_tkinter`. Однако вам в своих сценариях практически всегда придется импортировать модуль `tkinter`, а не `_tkinter` – последний, являясь модулем реализации, предназначен исключительно для внутреннего использования (и получил такое необычное название именно по этой причине).

Программная структура

К счастью, программистам на языке Python обычно не нужно беспокоиться обо всей этой интеграции и маршрутизации вызовов, реализуемыми внутренними механизмами: они просто создают виджеты и регистрируют функции на языке Python для обработки событий этих элементов. Однако из-за имеющейся в итоге структуры обработчики событий обычно называют обработчиками *обратного вызова*, потому что библиотека GUI «делает обратный вызов» программного кода на языке Python, когда происходят события.

Ниже мы увидим, что программы Python/tkinter полностью *управляются событиями*: они создают графические интерфейсы и регистрируют обработчики событий, а затем ничего не делают и только ждут, когда произойдут события. Во время этого ожидания библиотека Tk выполняет цикл событий, который следит за щелчками мыши, нажатием клавиш и так далее. Вся обработка, выполняемая прикладной программой, происходит в зарегистрированных обработчиках, вызываемых 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 к графическим интерфейсам – здесь нужно мыслить на языке небольших фрагментов программного кода.

В языке Python библиотека Tk становится *объектно-ориентированной* просто потому, что сам Python является объектно-ориентированным

языком: слой tkinter экспортирует прикладной интерфейс библиотеки Tk как классы Python. При работе с библиотекой tkinter можно использовать простой подход, основанный на вызовах функций, создающих виджеты и интерфейсы, или объектно-ориентированные приемы, такие как наследование и композиция, для настройки и расширения классов из базового набора tkinter. Большие графические интерфейсы на базе tkinter обычно строятся как деревья связанных между собой графических элементов и часто реализуются в виде классов Python, чтобы обеспечить структурированность и сохранность информации о состоянии в промежутках между событиями. В этой части книги мы увидим, что графические интерфейсы на базе tkinter, реализация которых представляется классами, почти по умолчанию становятся многократно используемыми программными компонентами.

Взбираясь по кривой обучения программированию графических интерфейсов

Теперь перейдем к программированию. Начнем с нескольких небольших примеров, иллюстрирующих основные понятия, и покажем, как создаются окна на экране компьютера. По ходу изложения примеры будут становиться более сложными, но для начала – об основных принципах.

«Hello World» в четыре строки (или меньше)

Обычно первым делом при изучении особенностей создания графических интерфейсов демонстрируется пример, выводящий в окне строку «Hello World». Пример 7.1 делает это в четырех строках.

Пример 7.1. PP4E\Gui\Intro\gui1.py

```
from tkinter import Label                      # импортировать виджет
widget = Label(None, text='Hello GUI world!') # создать его
widget.pack()                                  # разместить
widget.mainloop()                              # запустить цикл событий
```

Это законченная программа на языке Python, реализующая графический интерфейс с помощью библиотеки tkinter. При запуске этого сценария получается простое окно с меткой посередине. Его вид, как оно выглядит в Windows 7 на моем ноутбуке, показан на рис. 7.1 (я специально растягивал некоторые окна, изображения которых приводятся в этой книге, по горизонтали, чтобы заголовки окон были видны полностью; внешний вид окон у вас может немного отличаться, в зависимости от используемой платформы).

Пока здесь нечем особенно похвастать, но обратите внимание, что это независимое полнофункциональное окно на экране компьютера. Его можно распахнуть на весь экран, свернуть и спрятать на системной па-

нели, изменить его размер. Щелкните на кнопке «X» в правом верхнем углу окна, чтобы закрыть его и завершить программу.

Рис. 7.1. *«Hello World» (gui1) в Windows*

Кроме того, сценарий, создающий это окно, полностью переносим. Запустите его на своем компьютере, чтобы увидеть окно, создаваемое им. При запуске этого файла в Linux создается аналогичное окно, но ведет оно себя в соответствии с работающим в Linux менеджером окон. Даже в одной и той же системе один и тот же программный код на языке Python в разных оконных системах (например, в KDE и Gnome) может воспроизводить окна, выглядящие по-разному. Тот же сценарий будет воспроизводить окна, отличающиеся внешним видом, в Macintosh и других Unix-подобных системах. Однако на всех платформах его поведение будет одним и тем же.

Основы использования tkinter

Сценарий `gui1` является тривиальным примером, но он иллюстрирует шаги, которые выполняются большинством программ, использующих библиотеку tkinter. Этот сценарий делает следующее:

1. Загружает класс виджета из модуля `tkinter`.
2. Создает экземпляр импортированного класса `Label`.
3. Упаковывает (размещает) новый объект `Label` в его родительском элементе.
4. Вызывает функцию `mainloop`, чтобы показать окно и начать цикл событий tkinter.

Метод `mainloop`, вызываемый последним, помещает метку на экран и входит в состояние ожидания, в котором отслеживаются события графического интерфейса, генерируемые пользователем. Внутри функции `mainloop` библиотека tkinter следит за такими вещами, как клавиатура или мышь, чтобы обнаружить порожденные пользователем события. Функционально функция `mainloop` в библиотеке tkinter напоминает следующий псевдокод:

```
def mainloop():
    пока главное окно не закрыто:
        если возникло событие:
            вызвать соответствующий обработчик
```

В этой модели вызов `mainloop` в примере 7.1 никогда не вернет управление сценарию, пока графический интерфейс отображается на экране.[1] Как мы увидим, добравшись до больших сценариев, единственным способом выполнять какие-либо операции после вызова `mainloop` является регистрация обработчиков обратного вызова, реагирующих на события.

Это называется разработкой программ, *управляемых событиями*, и, возможно, это один из самых необычных аспектов графических интерфейсов. Программы с графическим интерфейсом принимают форму набора обработчиков событий, которые совместно используют хранящуюся информацию, а не линейного потока выполнения. Как это выглядит в действующем программном коде, мы увидим в последующих примерах.

Обратите внимание, что для создания графического интерфейса в этом сценарии действительно необходимо выполнить шаги 3 и 4. А чтобы отобразить окно, нужно вызвать `mainloop` – для вывода виджетов внутри окна они должны быть скомпонованы (то есть размещены), чтобы менеджер компоновки tkinter знал о них. На самом деле, если вызвать только `mainloop` или только `pack`, не вызывая второго из них, окно будет показывать не то, что нужно: `mainloop` без `pack` выведет пустое окно, а `pack` без `mainloop` не выведет ничего, потому что сценарий не войдет в состояние ожидания событий (можете попробовать). Иногда вызывать функцию `mainloop` необязательно, например, при программировании в интерактивной оболочке, но в общем случае вам не следует полагаться на это.

Поскольку понятия, иллюстрируемые этим простым примером, лежат в центре большинства программ tkinter, рассмотрим их несколько глубже, прежде чем двинуться дальше.

Создание виджетов

При создании графических элементов в tkinter можно указать, как они должны быть настроены. Сценарий `gui1` передает два аргумента конструктору класса `Label`:

- Первый аргумент определяет объект родительского виджета, к которому нужно прикрепить новую метку. В данном случае `None` означает «прикрепить новый виджет `Label` к установленному по умолчанию окну верхнего уровня данной программы». Позднее в этом аргумен-

[1] Формально вызов `mainloop` возвращает управление сценарию только после выхода из цикла событий. Обычно это происходит при закрытии главного окна, но может случиться и в ответ на явный вызов метода `quit`, который завершает вложенный цикл событий, но оставляет окно открытым. Почему это имеет значение, вы узнаете в главе 8.

те мы будем передавать фактические виджеты, чтобы прикреплять метки к другим объектам-контейнерам.

- Второй аргумент служит параметром настройки для `Label`. Он имеет форму именованного аргумента: параметр `text` определяет строку текста, которая должна появиться в виде метки. Большинство конструкторов виджетов принимают несколько именованных аргументов, определяющих различные параметры (цвет, размер, обработчики событий и так далее). Однако большинство параметров настройки виджетов имеют вполне разумные значения по умолчанию для каждой платформы, чем в значительной мере объясняется простота tkinter. Большинство параметров требуется определять, только если необходимо выполнить нестандартные настройки.

Как будет показано далее, аргумент, определяющий родительский виджет, является основой для построения сложных графических интерфейсов в виде деревьев виджетов. Библиотека tkinter действует по принципу «что построишь, то и получишь» – деревья объектов виджетов создаются как модели того, что мы хотим видеть на экране, а затем мы просим дерево изобразить себя с помощью вызова `mainloop`.

Менеджеры компоновки

Метод `pack` виджетов, к которому обращается сценарий `gui1`, вызывает менеджер компоновки – один из трех способов управления организацией графических элементов в окне. Менеджеры компоновки в библиотеке tkinter просто организуют один или несколько виджетов внутри контейнера (который иногда называется родителем или владельцем). Контейнерами могут служить окна верхнего уровня и фреймы (особая разновидность виджетов, с которой мы познакомимся далее), при этом сами контейнеры могут вкладываться внутрь других контейнеров, образуя иерархические отображения.

Для автоматического размещения виджетов в окне менеджер компоновки использует ограничивающие параметры. Сценарии передают высокоуровневые инструкции (например, «прикрепить данный виджет к верхней части контейнера и растянуть по вертикали до заполнения пространства»), а не абсолютные координаты в пикселях. Поскольку такие ограничения весьма абстрактны, менеджер компоновки предоставляет мощную и простую в использовании систему размещения элементов. В действительности вам даже не обязательно задавать ограничения – если не передавать аргументы методу `pack`, будет выполнено размещение по умолчанию, когда виджет прикрепляется к верхнему краю контейнера.

Мы неоднократно будем возвращаться к менеджеру компоновки в этой главе и использовать его во многих примерах книги. В главе 9 мы также познакомимся с альтернативным менеджером компоновки `grid` и системой размещения виджетов в контейнере в табличном виде (то есть по

строкам и колонкам). Третий вариант, менеджер компоновки `placer`, описывается в документации Tk и не описывается в данной книге – он менее популярен, чем менеджеры `pack` и `grid`, и его использование может оказаться непростым делом при создании крупных графических интерфейсов.

Запуск программ с графическим интерфейсом

Как и любой другой программный код на языке Python, модуль из примера 7.1 может быть запущен несколькими способами: путем запуска как самостоятельного сценария:

```
C:\...\PP4E\Gui\Intro> python gui1.py
```

с помощью операции импортирования из интерактивного сеанса Python или из другого файла модуля:

```
>>> import gui1
```

путем запуска его, как выполняемого файла Unix, если добавить в начало файла особую строку, начинающуюся с `#!`:

```
% gui1.py &
```

или любым другим способом, которым программы Python могут быть запущены на вашей платформе. Например, этот сценарий можно также запустить щелчком мыши на имени файла в проводнике Windows, или его код может быть введен в интерактивной оболочке, в ответ на приглашение `>>>`.[1] Можно даже выполнить его из программы на языке C, вызвав соответствующую функцию прикладного интерфейса встраивания (подробности об интеграции с программами на языке C можно найти в главе 20).

Иными словами, практически нет специальных правил, которым требуется следовать, чтобы запустить графический интерфейс, реализованный на языке Python. Интерфейс tkinter (и сама библиотека Tk) связан с интерпретатором Python. Когда программа на языке Python вызывает функции GUI, они просто за кулисами передаются встроенной графической подсистеме. Это облегчает написание инструментов командной строки, которые вызывают появление всплывающих окон, –

[1] Совет: Как уже предлагалось выше, при вводе программного кода, использующего библиотеку tkinter, в *интерактивной оболочке*, вам может не потребоваться вызывать функцию `mainloop`, чтобы отобразить виджеты. Это необходимо делать в текущей версии IDLE, но в простом интерактивном сеансе, запущенном в окне консоли, этого может не потребоваться. В любом случае управление будет возвращено интерактивной оболочке, как только вы закроете созданное окно. Примечательно, что если явно создать виджет главного окна вызовом конструктора `Tk()` и присоединить к нему виджеты (как описывается ниже), вам придется вызвать его снова после закрытия окна, иначе окно приложения не появится.

они выполняются так же, как обычные текстовые сценарии, которые изучались в предыдущей части книги.

Как избежать появления окна консоли DOS в Windows

В главах 3 и 6 отмечалось, что если имя программы имеет расширение *.pyw*, а не *.py*, запуск сценария Python в Windows не вызывает появления окна консоли DOS, которое обслуживает стандартные потоки ввода-вывода сценария, запускаемого щелчком мыши на ярлыке. Теперь, когда мы наконец-то начали создавать собственные окна, этот прием с именем файла становится еще более полезным.

Чтобы видеть только окна, создаваемые сценарием, независимо от способа его запуска, дайте файлам своих сценариев с графическим интерфейсом расширение *.pyw*, если они будут выполняться в Windows. Например, щелчок на файле примера 7.2 в проводнике Windows создает только окно, изображенное на рис. 7.1.

Пример 7.2. PP4E\Gui\Intro\gui1.pyw

```
...то же, что gui1.py...
```

Избежать появления всплывающих окон DOS в Windows можно также, запуская программу с выполняемым 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 – в Unix-подобных системах нет понятия всплывающих окон для подключения потоков ввода-вывода. С другой стороны, если в будущем потребуется выполнять ваши сценарии с графическим интерфейсом в Windows, добавление «w» в конце имени может освободить вас 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.

Альтернативные приемы использования tkinter

Как можно предполагать, есть разные способы реализации примера `gui1`. Например, если желательно более явным образом описать в сценарии импортируемые из tkinter элементы, импортируйте весь модуль и добавляйте имя модуля ко всем относящимся к нему именам, как в примере 7.3.

Пример 7.3. PP4E\Gui\Intro\gui1b.py – import против from

```
import tkinter
widget = tkinter.Label(None, text='Hello GUI world!')
widget.pack()
widget.mainloop()
```

В действующих примерах это может быть утомительным – библиотека tkinter экспортирует десятки классов графических переменных и кон-

стант, которые повсюду встречаются в сценариях с графическим интерфейсом на языке Python. На самом деле обычно проще использовать спецификатор *, чтобы одним махом импортировать все, находящееся в модуле tkinter. Это показано в примере 7.4.

Пример 7.4. PP4E\Gui\Intro\gui1c.py – константы и функции 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 необходимо, поэтому он один из немногих, для которых формат импорта со спецификатором * можно применять относительно безопасно.[1] К примеру, константа `TOP` в вызове функции `pack` является одной из многих, экспортируемых модулем `tkinter`. Это просто имя переменной (`TOP="top"`), которой заранее присвоено значение в модуле `constants`, автоматически загружаемом пакетом `tkinter`.

При размещении графических элементов можно указать, к какому краю родительского контейнера они должны быть прикреплены, – `TOP`, `BOTTOM`, `LEFT` или `RIGHT`. Если параметр `side` не передается функции `pack` (как в предшествующих примерах), виджет по умолчанию прикрепляется к верхнему краю (`TOP`). В целом, более крупные графические интерфейсы на базе tkinter могут конструироваться 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 размещения и параметром `side`. Если виджеты располагаются по сетке, им присваиваются номера строк и колонок. Однако все это имеет смысл, только если в окне присутствует больше одного виджета, поэтому продолжим наши исследования.

Обратите внимание, что в этой версии метод `pack` вызывается сразу после создания метки, которая не присваивается никакой переменной. Если нет необходимости сохранять виджет, можно разместить его таким способом на месте, и этим сэкономить строку программного кода. Мы будем использовать этот прием, если графический элемент прикрепляется к более крупной структуре и нигде в программном коде больше не используется. Сложности могут возникнуть, если присваивать

[1] Если рассмотреть главный файл tkinter в библиотеке исходного программного кода Python (в настоящее время *Lib\tkinter__init__.py*), 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 *. Модуль с константами называется *constants.py* и находится в том же самом каталоге пакета, хотя с течением времени имя и местоположение его может измениться.

результат, возвращаемый pack, но объяснение этого откладывается до того времени, когда будут изучены еще некоторые основные понятия.

Мы также используем здесь в качестве родителя экземпляр класса Tk виджета вместо None. Объект Tk представляет главное («корневое») окно программы – которое открывается вместе с запуском программы. Автоматически созданный объект Tk используется также как родительский виджет по умолчанию, когда в вызовы других методов виджетов не передается никакого родителя или когда в качестве родителя передается None. Иными словами, по умолчанию виджеты просто прикрепляются к главному окну программы. В данном сценарии это поведение по умолчанию реализуется явно, путем создания и передачи самого объекта Tk. В главе 8 будет показано, как для создания новых всплывающих окон, действующих независимо от главного окна программы, используются виджеты Toplevel.

Некоторые методы виджетов в tkinter экспортируются также в виде функций, что позволяет сократить пример 7.5 до трех строчек кода.

Пример 7.5. PP4E\Gui\Intro\gui1d.py – минимальная версия

```
from tkinter import *
Label(text='Hello GUI world!').pack()
mainloop()
```

Функцию mainloop из модуля tkinter можно вызывать относительно виджета или непосредственно (то есть как метод или как функцию). В этом варианте мы не передавали конструктору Label аргумент родителя: если он опущен, то по умолчанию будет использоваться значение None (что, в свою очередь, по умолчанию означает автоматически создаваемый объект Tk). Но использование этого значения по умолчанию становится менее удобным при создании более крупных графических интерфейсов. Такие элементы, как метки, чаще прикрепляются к другим контейнерам виджетов.

Основы изменения размеров виджетов

Размер окон верхнего уровня – таких как окно, создававшееся во всех продемонстрированных до сих пор вариантах, обычно может изменяться пользователем – нужно просто растянуть окно с помощью мыши. На рис. 7.2 показано, как выглядит увеличенное в размерах окно.

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 параметров метода pack, как показано в примере 7.6.

Пример 7.6. PP4E\Gui\Intro\gui1e.py – растягивание окна

```
from tkinter import *
Label(text='Hello GUI world!').pack(expand=YES, fill=BOTH)
mainloop()
```

Рис. 7.2. *Увеличенное окно сценария gui1*

При размещении виджетов можно указывать, должен ли виджет увеличиться в размерах, чтобы заполнить все свободное пространство, и если да, то как он должен измениться, чтобы заполнить все это пространство. По умолчанию виджеты не увеличиваются вслед за родителем. Но в данном сценарии имена `YES` и `BOTH` (импортированные из модуля tkinter) указывают, что метка должна увеличиваться вместе со своим родителем, то есть с главным окном. Так оно и происходит, как видно на рис. 7.3.

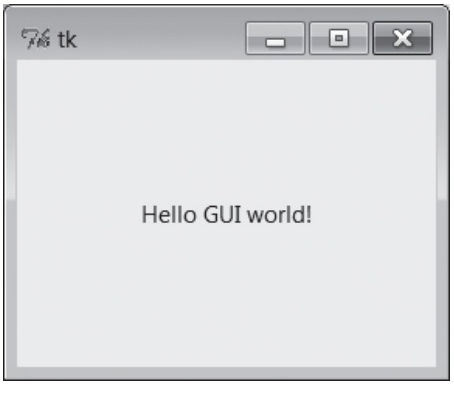


Рис. 7.3. *Сценарий gui1e с виджетом, изменяющим размер*

Технически, менеджер компоновки назначает размер каждому отображаемому виджету, исходя из его содержимого (длины текстовой строки и так далее). По умолчанию виджет может занимать только отведенное ему пространство и не может быть больше назначенного ему размера. Параметры `expand` и `fill` позволяют более точно определять правила автоматического изменения размеров:

Параметр `expand=YES`

Предлагает менеджеру компоновки выделять виджету все свободное пространство в родительском контейнере, не занятое другими виджетами.

Параметр `fill`

Может использоваться для растяжения виджета, чтобы он занял все выделенное ему пространство.

Сочетания этих двух параметров производят различные эффекты расположения и изменения размеров, при этом некоторые из них имеют смысл только при наличии в окне нескольких графических элементов. Например, использование параметра `expand` без `fill` приводит к размещению виджета в центре занимаемого им пространства, а параметр `fill` может определять возможность изменения размеров только по вертикали (`fill=Y`), только по горизонтали (`fill=X`) или по обеим осям одновременно (`fill=BOTH`). Определяя эти ограничения и стороны прикрепления для всех виджетов в графическом интерфейсе наряду с порядком размещения, можно довольно точно управлять их взаимным расположением. В последующих главах будет показано, что менеджер компоновки `grid` использует совершенно другой протокол изменения размеров, но при необходимости может обеспечивать похожий порядок размещения.

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 параметров `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 «изменить размеры отведенного мне места, чтобы оно занимало все свободное пространство, и растянуть меня так, чтобы заполнить освободившееся 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 библиотеке tkinter, что должно выводиться в метке, передавая ее текст в именованном аргументе при вызове конструктора метки. Оказывается, существует еще два способа определения параметров метки. В примере 7.7 параметр `text` метки определяется после ее создания путем присвоения ключу `text` виджета – виджеты перегружают операции индексирования, чтобы доступ к параметром можно было осуществлять по ключам, подобно словарям.

Пример 7.7. PP4E\Gui\Intro\gui1f.py – параметры

```
from tkinter import *
widget = Label()
widget['text'] = 'Hello GUI world!'
```

```
widget.pack(side=TOP)
mainloop()
```

Но чаще параметры виджетов устанавливаются после их создания путем вызова метода `config`, как в примере 7.8.

Пример 7.8. PP4E\Gui\Intro\gui1g.py – методы config и title

```
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` (допускается также использовать его синоним `configure`) можно вызвать в любой момент после создания виджета, чтобы на лету изменить его внешний вид. Например, можно было бы вызвать метод `config` еще раз, дальше в сценарии, чтобы изменить текст, который выводится в метке. Примеры такого динамического изменения параметров можно найти в последующих примерах в этой части книги.

Обратите также внимание, что в этой версии вызывается метод `root.title` – этот метод устанавливает текст в заголовке окна, как показано на рис. 7.4. Вообще говоря, окна верхнего уровня, такие как виджет `root` типа `Tk`, в этом примере экспортируют интерфейсы менеджера окон, имеющие отношение к рамке окна, а не к его содержимому.

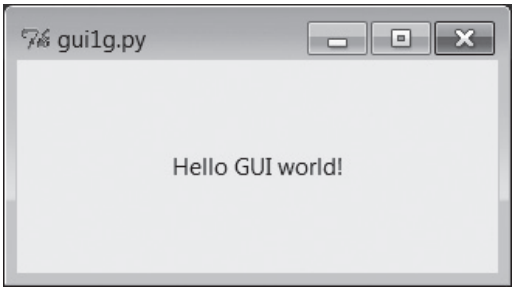


***Рис. 7.4.** Сценарий gui1g с виджетом, изменяющим размер, и заголовком окна*

Развлечения ради в этой версии метка также центрируется при изменении размеров окна путем установки значений параметров `expand` и `fill`. На самом деле здесь почти все делается явно, и это отражает типичный способ создания меток в полноценных интерфейсах – с указанием родителя, политики автоматического изменения размеров и способа прикрепления, без использования значений по умолчанию.

Еще одна версия в память о былых временах

Наконец, если вы склонны к минимализму и испытываете ностальгию по старому стилю программирования на языке Python, сценарий «Hello World» можно записать, как показано в примере 7.9.

Пример 7.9. PP4E\Gui\Intro\gui1-old.py — использование словаря

```
from tkinter import *
Label(None, {'text': 'Hello GUI world!', Pack: {'side': 'top'}}).mainloop()
```

В этом примере для создания окна хватило двух строк, хотя выглядит он ужасно! Эта схема основана на старом стиле программирования, широко использовавшемся до выхода версии Python 1.3, когда параметры настройки передавались не как именованные аргументы, а в виде словаря.[1] В этой схеме параметры менеджера компоновки могут передаваться как значения по ключу `Pack` (имя класса в модуле `tkinter`).

Схема вызова с использованием словаря по-прежнему действует, и ее можно обнаружить в старых сценариях на языке Python, но лучше ею не пользоваться. Для передачи параметров применяйте в своих сценариях, использующих tkinter, именованные аргументы и явный вызов метода `pack`. Единственная причина, по которой я не выкинул этот пример, заключается в том, что словари все еще могут быть полезны, когда требуется динамически вычислить и передать набор параметров.

С другой стороны, теперь можно использовать синтаксическую конструкцию `func(*pargs, **kargs)`, которая также позволяет передать явный словарь именованных аргументов в третьей позиции:

```
options = {'text': 'Hello GUI world!'}
layout = {'side': 'top'}
Label(None, **options).pack(**layout) # ключи должны быть строками
```

Даже в динамических сценариях, где параметры виджетов определяются во время выполнения, нет веских причин использовать формат вызова со словарем, который использовался с версиями, предшествовавшими tkinter 1.3.

Добавление виджетов без их сохранения

В сценарии *gui1c.py* из примера 7.4 я начал добавлять метки, не сохраняя их в переменных. Этот прием действует и является совершенно допустимым, но поскольку при первом взгляде такой программный код

[1] В действительности именованные аргументы впервые появились в Python, чтобы сделать более очевидными такие вызовы tkinter. Внутри интерпретатора именованные аргументы передаются как словарь (который можно получить, указав аргумент вида `**name` в инструкции `def` объявления функции), поэтому обе схемы аналогичны по реализации. Но они отличаются количеством символов, которые нужно ввести и проверить.

может смутить начинающих программистов, необходимо более подробно объяснить здесь, почему он работает.

В tkinter объекты классов Python соответствуют 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 соответствующего ему объекта на экране.

К счастью, сценариям на языке Python обычно не требуется беспокоиться о времени жизни объекта. Обычно они вообще не обязаны сохранять ссылки на создаваемые объекты виджетов, если только не собираются в дальнейшем изменять параметры настройки этих объектов. Например, при программировании с использованием tkinter часто сразу же после создания виджет размещают в интерфейсе, если в дальнейшем ссылка на него не понадобится:

```
Label(text='hi').pack()                # OK
```

Эта инструкция выполняется как обычно, слева направо. Она сначала 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 `Label` является временным: так как он не присваивается переменной, в обычной ситуации он должен был бы быть утилизирован сборщиком мусора сразу после выполнения метода `pack`.

Однако поскольку при создании объектов библиотека tkinter обращается к библиотеке Tk, метка будет нарисована на экране, как положено, несмотря на то, что мы не сохранили в своем сценарии ссылку на соответствующий объект Python. В действительности внутри tkinter объекты виджетов связываются в долгоживущее дерево, которое используется для представления интерфейса, отображаемого на экране, поэтому ссылка на объект `Label`, созданный этой инструкцией, сохраняется, хотя и не внутри нашего сценария.[1]

---

[1] Бывшим программистам на языке Tcl, возможно, будет интересно узнать, что Python не только внутренне строит дерево виджетов, но и использует его для автоматического создания строк путей к виджетам, которые приходится вручную писать в Tcl/Tk (например, `.panel.row.cmd`). Python использует адреса объектов классов виджетов, помещает их в компоненты пути и сохраняет полученные пути в дереве виджетов. Например, метке, прикрепленной к контейнеру, может быть присвоено внутреннее имя, такое как `.8220096.8219408`. Однако вас это не должно волновать. Просто создавайте и связывайте объекты графических элементов, передавая их родителей, и предоставьте Python самому работать с путями, опираясь на дерево объектов. Дополнительные сведения об отношениях Tk и tkinter приводятся в конце главы.

Иными словами, сценариям обычно не нужно беспокоиться о сроке жизни объектов виджетов, и вполне допустимо создавать виджеты и сразу же размещать их одной и той же инструкцией, не сохраняя ссылки на них в переменных.

Но это не значит, что можно писать такой код:

```
widget = Label(text='hi').pack()        # неверно!
...использование графического элемента...
```

На первый взгляд, эта инструкция должна присвоить только что размещенную в интерфейсе метку переменной `widget`, но это не так. На самом деле это известная ошибка новичков в tkinter. Метод `pack` размещает виджет, но не возвращает его. Метод `pack` возвращает объект `None` – после этой операции переменная `widget` будет хранить ссылку на `None` и все дальнейшие операции с виджетом, использующие эту переменную, окажутся безуспешными. Например, по той же причине следующая команда потерпит неудачу:

```
Label(text='hi').pack().mainloop()      # неверно!
```

Метод `pack` возвращает `None`, поэтому попытка обратиться к его атрибуту `mainloop` возбудит исключение (как и должно быть). Если вы действительно хотите разместить виджет и сохранить ссылку на него, используйте такой способ:

```
widget = Label(text='hi')               # тоже правильно
widget.pack()
...использование графического элемента...
```

Такой способ несколько многословнее, но он более надежен, чем размещение виджета в той же инструкции, которая его создает, и позволяет сохранить ссылку на виджет для последующих операций с ним. Кроме того, такой способ чаще используется на практике при создании более сложных конфигураций и схем размещения виджетов.

С другой стороны, сценарии, занимающиеся компоновкой внешнего вида, часто добавляют виджеты, размещая их раз и навсегда и не нуждаясь в последующей их настройке, – сохранение долгоживущих ссылок в такой программе не имеет смысла и является необязательным.

В главе 8 мы встретимся с двумя исключениями из этого правила. Сценарии должны вручную получать ссылки на объекты *изображений*, потому что фактические данные, составляющие изображение, будут уничтожены, как только сборщик мусора в Python утилизирует объект изображения. Объекты класса `Variable` из библиотеки tkinter также временно сбрасывают связанные с ними переменные из библиотеки Tk при утилизации, но это случается редко и не так опасно.

Добавление кнопок и обработчиков

Пока мы научились только выводить текст в метках и попутно познакомились с базовыми понятиями tkinter. Метки хороши для обучения основам, но от пользовательских интерфейсов обычно требуется нечто большее – способность реагировать на действия пользователя. Эту способность демонстрирует программа из примера 7.10, которая создает окно, изображенное на рис. 7.5.

Пример 7-10. PP4E\Gui\Intro\gui2.py

```
import sys
from tkinter import *
widget = Button(None, text='Hello widget world', command=sys.exit)
widget.pack()
widget.mainloop()
```

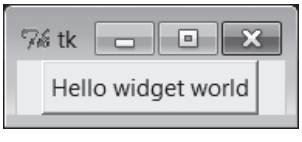


***Рис. 7.5.** Кнопка вверху*

Здесь мы создаем вместо метки экземпляр класса `Button`. Как и прежде, он прикрепляется по умолчанию к верхнему краю `TOP` окна верхнего уровня. Но главное, на что нужно обратить внимание, это аргументы с параметрами кнопки: в качестве значения параметра с именем `command` используется функция `sys.exit`.

В кнопках параметр `command` определяет функцию обратного вызова, которая должна вызываться в дальнейшем при нажатии кнопки. Фактически с помощью параметра `command` регистрируется действие, которое должна вызвать библиотека tkinter, когда в виджете произойдет событие. Функция обратного вызова, использованная здесь, не представляет большого интереса: как мы узнали в главе 5, 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 в случае с метками, есть разные способы создания кнопок. Пример 7.11 является версией, в которой кнопка размещается в интерфейсе немедленно, без сохранения ссылки на нее в переменной. Она явно прикрепляется к левому краю родительского окна, и в качестве обработчика для нее определяется `root.quit` – стандартный метод объекта `Tk`, закрывающий главное окно и тем самым завершающий программу. В действительности метод `quit` завершает выполнение функции `mainloop` цикла событий и всей программы в целом – когда мы начнем использовать сразу несколько окон верхнего уровня в главе 8, мы

узнаем, что метод quit обычно закрывает все окна, а родственный ему метод destroy закрывает только одно окно.

Пример 7.11. PP4E\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, изображенное на рис. 7.6. Мы не потребовали от кнопки, чтобы она автоматически изменяла свои размеры и занимала все свободное пространство, – она этого и не делает.

***Рис. 7.6.** Кнопка слева*

В двух последних примерах нажатие на кнопку завершает выполнение программы. В более старых сценариях, использующих tkinter, иногда можно увидеть, как параметру command присваивается строка exit, что также вызывает закрытие окна при нажатии на кнопку. В этом случае используется инструмент, имеющийся в библиотеке Tk, но такой прием менее характерен для Python, чем sys.exit или root.quit.

Еще раз об изменении размеров виджетов: растягивание

Даже для такого простого интерфейса есть множество способов определить его внешний вид с помощью основанного на ограничениях менеджера компоновки pack. Например, чтобы поместить кнопку в центре окна, добавьте в вызов метода pack параметр expand=YES в примере 7.11, как показано ниже:

```
Button(root, text='press', command=root.quit).pack(side=LEFT, expand=YES)
```

В этом случае менеджер компоновки отдаст кнопке все свободное пространство, но не растянет ее. В результате получится окно, как на рис. 7.7.

***Рис. 7.7.** pack(side=LEFT, expand=YES)*

Если необходимо, чтобы кнопке было отдано все свободное пространство и она была растянута по горизонтали, добавьте в вызов pack име-

нованные аргументы `expand=YES` и `fill=X`. В результате получится то, что изображено на рис. 7.8.

Рис. 7.8. *pack(side=LEFT, expand=YES, fill=X)*

Кнопка первоначально займет все окно (выделенное ей место расширено, а сама она растянута, чтобы заполнить выделенное пространство). При этом кнопка будет растягиваться с увеличением размеров родительского окна. Как показано на рис. 7.9, кнопка в этом окне будет растягиваться при растягивании родителя, но только по горизонтальной оси X.

Рис. 7.9. *Изменение размера при expand=YES, fill=X*

Чтобы кнопка растягивалась в обоих направлениях, укажите в вызове `pack` параметры `expand=YES` и `fill=BOTH` – теперь при изменении размеров окна кнопка будет растягиваться во все стороны, как показано на рис. 7.10. Если раскрыть окно на весь экран, получится одна очень большая кнопка tkinter.

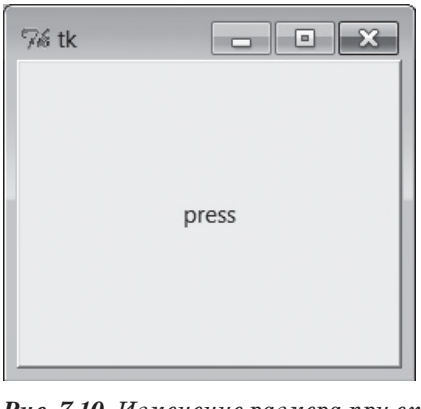


Рис. 7.10. *Изменение размера при expand=YES, fill=BOTH*

В более сложных интерфейсах такая кнопка будет растягиваться, только когда автоматическое изменение размеров задано для всех содержащих ее виджетов. Здесь единственным родителем кнопки является корневое окно `Tk` программы, поэтому вопрос об автоматическом изменении размеров родителя пока не стоит – в последующих сценариях нам потребуется также делать автоматически расширяемыми объемлющие виджеты `Frame`. Мы еще вернемся к менеджеру компоновки, когда встретимся с интерфейсами, содержащими несколько виджетов, и еще раз – когда будем изучать альтернативную функцию `grid` в главе 9.

Добавление пользовательских обработчиков

В простых примерах с кнопкой из предыдущего раздела обработчиком событий служила уже существующая функция, завершающая программу. Однако ничуть не сложнее зарегистрировать собственный обработчик, который выполняет что-нибудь более полезное. В примере 7.12 определяется собственный обработчик обратного вызова на языке Python.

Пример 7.12. PP4E\Gui\Intro\gui3.py

```
import sys
from tkinter import *

def quit():                              # собственный обработчик событий
    print('Hello, I must be going...')
    sys.exit()                           # закрыть окно и завершить процесс

widget = Button(None, text='Hello event world', command=quit)
widget.pack()
widget.mainloop()
```

Окно, создаваемое этим сценарием, изображено на рис. 7.11. Этот сценарий и воспроизводимый им графический интерфейс почти совпадают с предыдущим примером. Но здесь в параметре `command` передается функция, которая определена локально. При нажатии кнопки библиотека tkinter вызовет для обработки события функцию `quit` в этом файле. Внутри `quit` вызов функции `print` выведет сообщение в поток `stdout` программы, и процесс завершится, как ранее.

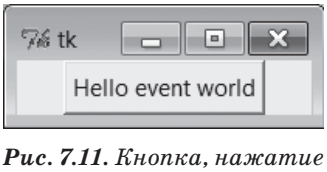


***Рис. 7.11.** Кнопка, нажатие на которую вызывает функцию Python*

Как обычно, роль потока вывода `stdout` играет окно, из которого была запущена программа, если только он не был перенаправлен в файл.

Это будет всплывающее окно консоли DOS, если запустить программу щелчком на файле в Windows; добавьте вызов `input` перед `sys.exit`, если всплывающее окно исчезает прежде чем удается посмотреть сообщение. Ниже показано, как выглядит вывод в стандартный поток вывода при нажатии кнопки – он генерируется функцией на языке Python, которую автоматически вызывает библиотека tkinter:

```
C:\...\PP4E\Gui\Intro> python gui3.py
Hello, I must be going...

C:\...\PP4E\Gui\Intro>
```

Обычно такие сообщения должны выводиться в графическом интерфейсе, но мы еще не добрались до того, как это сделать. Конечно, функции обратного вызова обычно выполняют больше действий (и могут даже вызывать появление новых независимых окон), но основы их использования продемонстрированы в данном примере.

Вообще говоря, обработчиками событий могут быть любые вызываемые объекты – функции; анонимные функции, создаваемые с помощью `lambda`-выражений; связанные методы классов или экземпляров типов или экземпляры классов, наследующие метод перегрузки оператора `__call__`. Обработчики нажатия кнопки `Button` не получают никаких аргументов (кроме `self`, который автоматически передается связанным методам) – любые данные, необходимые обработчику события, должны быть доступны другими способами: в виде глобальных переменных, атрибутов экземпляра класса, дополнительных аргументов, передаваемых косвенным путем, и так далее.

lambda-выражения как обработчики событий

Напомню, что `lambda`-выражения в языке Python при выполнении генерируют новые объекты анонимных функций. Если необходимо, чтобы в функцию обработчика передавались дополнительные данные, можно зарегистрировать `lambda`-выражение, откладывающее вызов фактической функции, и определить в нем необходимые дополнительные данные.

Как этим воспользоваться, будет показано далее в этой части книги, но для иллюстрации основных идей в примере 7.13 демонстрируется, как будет выглядеть приложение 7.12, если задействовать в нем инструкцию `lambda` вместо `def`.

Пример 7.13. PP4E\Gui\Intro\gui3b.py

```
import sys
from tkinter import *          # lambda-выражение генерирует функцию

widget = Button(None,          # но содержит всего лишь выражение
            text='Hello event world',
            command=(lambda: print('Hello lambda world') or sys.exit()) )
```

```
widget.pack()
widget.mainloop()
```

В этом примере есть небольшая хитрость: lambda-выражения могут содержать только одно выражение, поэтому здесь используется оператор or, чтобы заставить выполниться два выражения (функция print вызывается первой, а поскольку в Python 3.X она стала функцией, нам не требуется использовать sys.stdout непосредственно).

Отложенные вызовы с применением инструкций lambda и ссылок на объекты

Очень часто lambda-выражения используются для передачи дополнительных данных в обработчик события (для простоты я опустил вызовы функций pack и mainloop в следующем фрагменте):

```
def handler(A, B): # обычно вызывается без аргументов
    ...использование A и B...

X = 42
Button(text='ni', command=(lambda: handler(X, 'spam'))) # lambda добавляет
                                                        # аргументы
```

Библиотека tkinter вызывает обработчики command, не передавая им никаких аргументов, тем не менее с помощью такого lambda-выражения можно создать косвенную анонимную функцию, которая будет играть роль оболочки для действительного обработчика и передавать ему информацию, существовавшую в момент создания графического интерфейса. Вызов фактического обработчика *откладывается*, благодаря чему мы получаем возможность добавлять необходимые аргументы. В данном случае значение глобальной переменной X и строка "spam" будут переданы в аргументах A и B даже при том, что библиотека tkinter вызывает функции обратного вызова без аргументов. Таким образом, инструкция lambda может использоваться для отображения вызова без аргументов в вызов с аргументами, которые поставляются самим lambda-выражением.

Если такой синтаксис приводит вас в замешательство, запомните, что такое lambda-выражение, как показано выше, обычно может быть реализовано в виде простой инструкции def. Вторая функция в следующем фрагменте выполняет ту же операцию, что и lambda-выражение в предыдущем фрагменте, – ссылка на нее в операции создания кнопки фактически откладывает вызов действительного обработчика, чтобы ему можно было передать дополнительные аргументы:

```
def handler(A, B): # обычно вызывается без аргументов
    ...использование A и B...

X = 42
def func(): # косвенная функция-обертка, добавляющая аргументы
```

```
        handler(X, 'spam')

    Button(text='ni', command=func)
```

Чтобы более отчетливо понять, зачем необходимы отложенные вызовы, обратите внимание, что произойдет, если в операцию создания кнопки вставить вызов самого обработчика с аргументами без использования lambda-выражения или другой промежуточной функции, – обработчик будет вызван немедленно, *в момент создания кнопки*, а не когда на ней будет выполнен щелчок. Именно для того чтобы отложить вызов обработчика, обращение к нему требуется обернуть промежуточной функцией:

```
def handler(name):
    print(name)

Button(command=handler('spam')) # ОШИБКА: обработчик будет вызван немедленно!
```

Использование lambda-выражений или ссылок на вызываемые объекты позволяет отложить вызов обработчика до появления события. Например, использование lambda-выражения для передачи дополнительных данных с помощью определения встроенной функции, вызов которой откладывается:

```
def handler(name):
    print(name)

Button(command=(lambda: handler('spam'))) # OK: обертывание lambda-выражением
                                          #     откладывает вызов
```

всегда эквивалентно более длинной и, по мнению некоторых, менее удобной форме с двумя функциями:

```
def handler(name):
    print(name)

def temp():
    handler('spam')

Button(command=temp) # OK: ссылка на функцию, а не ее вызов
```

Нам достаточно применять только какой-то один прием – либо lambda-выражение без аргументов, либо ссылку на вызываемый объект, не принимающий аргументов, но *не оба* сразу. Бессмысленно использовать lambda-выражение, просто вызывающее функцию, которая не принимает дополнительных аргументов, так как в этом случае будет выполняться один лишний вызов:

```
def handler(name):
    print(name)

def temp():
```

```
    handler('spam')

Button(command=(lambda: temp())) # БЕССМЫСЛЕННО: добавляет лишний вызов!
```

Как будет показано далее, допускается также использовать ссылки на другие вызываемые объекты, такие как связанные методы и вызываемые экземпляры классов, которые сохраняют необходимую информацию в своих атрибутах. Если они не принимают аргументов, их имена можно просто указывать на этапе создания виджетов и их не требуется обертывать лишними `lambda`-выражениями.

Проблемы с областями видимости обработчиков

Использование `lambda`-выражения или промежуточной функции позволяет отложить вызов обработчика и передать ему дополнительные аргументы, тем не менее эти приемы влекут за собой некоторые проблемы, связанные с областью видимости, незаметные на первый взгляд. Это сфера базовых особенностей языка, но данные проблемы довольно часто встречаются на практике при создании графических интерфейсов.

Аргументы и глобальные переменные

Например, обратите внимание, что первая версия обработчика из предыдущего раздела сама могла бы обратиться к переменной `X` непосредственно, так как она является глобальной (и будет определена к моменту, когда будет вызван обработчик). Благодаря этому мы могли бы написать обработчик, принимающий единственный аргумент, и передавать ему в `lambda`-выражении только строку `'spam'`:

```
def handler(A): # X находится в глобальной области видимости
    ...использование глобальной переменной X и аргумента A...

X = 42
Button(text='ni', command=(lambda: handler('spam')))
```

Аргумент `A` также можно было бы перенести в глобальную область видимости, чтобы полностью устранить необходимость использования `lambda`-выражения; в этом случае мы сможем регистрировать сам обработчик и убрать промежуточную функцию.

Несмотря на всю простоту этого тривиального примера, на практике предпочтительнее использовать аргументы, а не глобальные переменные, потому что они делают внешние зависимости более очевидными, благодаря чему программный код проще будет понять и изменить. Один и тот же обработчик мог бы использоваться и в других контекстах, если не привязывать его к глобальным переменным. Пока мы не приступили к изучению крупных примеров, предъявляющих более сложные требования к сохранению информации о состоянии, вам придется принять на веру, что отказ от использования глобальных переменных в функ-

циях обратного вызова и в графическом интерфейсе в целом, делает их более доступными для повторного использования и обеспечивает поддержку нескольких экземпляров в одной и той же программе. Причем такой подход рекомендуется применять не только в графических интерфейсах.

Передача значений из объемлющей области видимости с помощью аргументов со значениями по умолчанию

Еще более тонкая проблема: обратите внимание, что если создание кнопки в этом примере будет выполняться внутри функции, а не на верхнем уровне сценария, имя `X` перестанет быть глобальным и окажется в локальной области видимости объемлющей функции. Может создаться ощущение, что переменная `X` исчезнет сразу после выхода из функции, еще до того, как возникнет событие, которое вызовет выполнение `lambda`-выражения:

```
def handler(A, B):
    ...использование A и B...

def makegui():
    X = 42
    Button(text='ni', command=(lambda: handler(X, 'spam'))) # запоминает X

makegui()
mainloop() # в этой точке область видимости функции makegui прекратила
           # существование
```

К счастью, модель ссылок в объемлющую область видимости, реализованная в Python, автоматически сохранит значение `X`, находящееся в локальной области видимости объемлющей функции, где используется `lambda`-выражение, и обеспечит его доступность в обработчике, когда он будет вызван по событию нажатия кнопки. Обычно этот прием действует именно так, как нам хотелось бы, и автоматически обслуживает подобные ссылки на переменные.

Однако такое использование объемлющей области видимости можно сделать более явным, применив аргументы со значениями по умолчанию, которые также сохраняют значения переменных из локальной области видимости объемлющей функции после выхода из нее. Например, в следующем фрагменте аргумент с именем `X` (слева в выражении `X=X`) сохранит объект `42`, потому что переменная с именем `X` (справа в выражении `X=X`) используется не только в объемлющей области видимости, но и в сгенерированной функции, которая позднее будет вызываться без аргументов:

```
def handler(A, B):     # аргументы со значениями по умолчанию обеспечивают
    ...использование A и B...  # сохранение информации о состоянии
```

```
def makegui():
    X = 42
    Button(text='ni', command=(lambda X=X: handler(X, 'spam')))
```

Так как значения по умолчанию аргументов вычисляются и сохраняются на этапе выполнения инструкции `lambda` (а не на этапе вызова функции, сгенерированной этой инструкцией), они обеспечивают явный способ сохранения объектов, доступ к которым будет осуществляться уже в процессе обработки события. Позднее библиотека вызовет `lambda`-функцию без аргументов, поэтому все ее аргументы получат значения по умолчанию.

В оригинальной версии этого примера не было никаких проблем, потому что переменная `X` находилась в глобальной области видимости и потому была доступна функции, сгенерированной `lambda`-выражением. Однако при использовании внутри функции переменная `X` может исчезнуть после выхода из объемлющей функции.

Передача значений из объемлющей области видимости с помощью автоматических ссылок

Аргументы со значениями по умолчанию способны более четко обозначить внешние зависимости, тем не менее можно обойтись и без них (начиная с версии Python 2.2, по крайней мере), и в современной практике они обычно не используются для этих целей. Вместо этого просто создается `lambda`-выражение, которое откладывает вызов действительного обработчика и передает ему дополнительные аргументы. Переменные из объемлющей области видимости, используемые в `lambda`-выражении, автоматически сохраняются даже после выхода из объемлющей функции.

Предыдущий фрагмент, например, в наши дни можно записать, как мы уже делали это выше — имя `X` в обработчике автоматически будет отображено в переменную `X` из объемлющей области видимости, что фактически приводит к сохранению значения, которое имела переменная `X` на момент создания кнопки:

```
def makegui():                                          # X запоминается
    X = 42                                              # автоматически
    Button(text='ni', command=(lambda: handler(X, 'spam'))) # аргументы по
                                                        # умолчанию не
                                                        # нужны
```

Далее мы увидим более конкретные примеры использования этого приема. При использовании классов для создания графического интерфейса, например, аргумент `self` является для методов локальной переменной и потому автоматически становится доступным в теле `lambda`-выражения. Нет никакой необходимости передавать его явно, в виде аргумента со значением по умолчанию:

```
class Gui:
    def handler(self, A, B):
        ...использование self, A и B...
    def makegui(self):
        X = 42
        Button(text='ni', command=(lambda: self.handler(X, 'spam')))

Gui().makegui()
mainloop()
```

Однако при использовании классов в качестве источника дополнительной информации для функций обратного вызова можно использовать атрибуты экземпляров и отказаться от дополнительных аргументов. Как это делается, мы увидим чуть ниже. Но сначала нам следует отклониться от темы и рассмотреть правила областей видимости в Python, чтобы понять, почему аргументы со значениями по умолчанию иногда все же бывают необходимы для передачи значений во вложенные `lambda`-функции, особенно в графических интерфейсах.

Но иногда вместо объемлющих областей видимости все же необходимо использовать аргументы со значениями по умолчанию

В давно написанных сценариях Python все еще можно увидеть, как аргументы со значениями по умолчанию используются для передачи ссылок в объемлющую область видимости, но в настоящее время предпочтение отдается приему автоматического получения ссылок. Может сложиться впечатление, что новые правила поиска во вложенных областях видимости, реализованные в Python, автоматизируют и делают ненужной ручную операцию передачи значений из объемлющей области видимости в виде аргументов со значениями по умолчанию.

Все это так, но не совсем. Есть одна особенность. Оказывается, что внутри `lambda`-выражения (или внутри инструкции `def`) ссылки на переменные в объемлющей области видимости разрешаются в момент *вызова* сгенерированной функции, а не в момент ее создания. Вследствие этого, когда функция будет вызвана позднее, такие ссылки будут отражать последние значения переменных, присвоенные им где бы то ни было в объемлющей области видимости, которые не обязательно будут совпадать со значениями, присвоенными переменным на момент создания функции. Это справедливо не только для вложенных функций обратного вызова, но и для находящихся в глобальной области видимости модуля. В любом случае, все ссылки на переменные в объемлющей области видимости разрешаются в момент вызова функции, а не в момент ее создания.

Аргументы со значениями по умолчанию отличаются тем, что их значения определяются в момент *создания* функции, а не в момент ее вы-

зова. Вследствие этого они по-прежнему могут быть полезным способом сохранения значений переменных из объемлющей области видимости, имевших место на момент создания функции. В отличие от ссылок на переменные в объемлющей области видимости, аргументы со значениями по умолчанию не получат другое значение, даже если переменная в объемлющей области видимости изменится между моментами создания и вызова функции. (Фактически именно этим объясняется, почему изменяемые объекты, такие как списки, передаваемые по умолчанию, сохраняются между вызовами – они создаются только один раз, в момент создания функции, и присоединяются к самой функции.)

Обычно это отличие не вызывает проблем, потому что в большинстве случаев используются ссылки на переменные в объемлющей области видимости, значения которым присваиваются только один раз (примером может служить аргумент `self` в методах классов). Но при непонимании этого отличия можно допустить программные ошибки, особенно если функции создаются в *цикле,* – при использовании в этих функциях ссылок на переменную цикла все они будут получать значение, полученное в *последней* итерации. Напротив, при использовании значений по умолчанию каждая функция будет получать *текущее* значение переменной цикла, а не последнее.

Из-за этого отличия одних только ссылок на переменные в объемлющей области видимости может оказаться недостаточно для сохранения значений, и аргументы со значениями по умолчанию все еще могут оказаться востребованными. Посмотрим, что это означает с точки зрения программного кода. Взгляните на следующую вложенную функцию (фрагменты программного кода для этого раздела вы найдете в файле *defaults.py* в дереве примеров на случай, если у вас появится желание поэкспериментировать с ними).

```
def simple():
    spam = 'ni'
    def action():
        print(spam) # ссылка на переменную в объемлющей функции
    return action

act = simple()      # создать и вернуть вложенную функцию
act()               # затем вызвать ее: выведет 'ni'
```

Это простейший случай ссылки на переменную в объемлющей области видимости, и он одинаково действует для вложенных функций, сгенерированных с помощью инструкций `def` и `lambda`. Но обратите внимание, что он будет действовать точно так же, если значение переменной `spam` будет присвоено *после* определения вложенной функции:

```
def normal():
    def action():
        return spam # поиск переменной будет выполняться в момент вызова
    spam = 'ni'
```

```
    return action

act = normal()
print(act())        # также выведет 'ni'
```

Из этого примера следует, что разрешение имен в объемлющей области видимости не выполняется в момент создания вложенной функции – фактически в этот момент переменная вообще не была определена. Разрешение имени выполняется в момент вызова вложенной функции. То же справедливо и для lambda-выражений:

```
def weird():
    spam = 42
    return (lambda: spam * 2) # запомнит ссылку на spam в объемлющей
                              # области видимости
act = weird()
print(act())                  # выведет 84
```

Пока все неплохо. Ссылка на переменную spam внутри вложенной lambda-функции сохраняет значение, полученное переменной в объемлющей области видимости, которое остается доступным даже после выхода из объемлющей функции. Этот пример соответствует зарегистрированному обработчику в графическом интерфейсе, который будет вызываться позднее, по событиям. И снова ссылка из вложенной области видимости разрешается не в тот момент, когда в lambda-выражение создает функцию, а когда сгенерированная функция будет *вызвана*. Следующий фрагмент более наглядно демонстрирует действие этого правила:

```
def weird():
    tmp = (lambda: spam * 2) # запоминает ссылку на spam, даже при том,
    spam = 42                # что здесь она еще не установлена
    return tmp

act = weird()
print(act())                 # выведет 84
```

Здесь вложенная функция ссылается на переменную, которой на момент создания функции еще не было присвоено значение. Действительно, при каждом вызове вложенной функции ссылки в объемлющую область видимости возвращают последние значения, присвоенные переменным. Взгляните, что происходит в следующем фрагменте:

```
def weird():
    spam = 42
    handler = (lambda: spam * 2) # функция не сохраняет текущее значение 42
    spam = 50
    print(handler()) # выведет 100: поиск spam выполняется именно сейчас
    spam = 60
    print(handler()) # выведет 120: поиск spam снова выполняется именно сейчас

weird()
```

Теперь значение ссылки на `spam` внутри `lambda`-функции изменяется при каждом вызове сгенерированной функции! Фактически ссылка на переменную возвращает *последнее* значение, присвоенное в объемлющей области видимости на момент вызова вложенной функции, потому что ссылки разрешаются в момент вызова функции, а не в момент ее создания.

С точки зрения графических интерфейсов, это становится существенным чаще всего, когда функции обработчиков генерируются в циклах и при этом используются ссылки в объемлющую область видимости для сохранения дополнительных данных, создаваемых внутри циклов. Если вы собираетесь генерировать функции внутри цикла, вы должны учитывать, что продемонстрированные выше особенности поведения применяются и к переменной цикла:

```
def odd():
    funcs = []
    for c in 'abcdefg':
        funcs.append((lambda: c)) # поиск переменной c будет выполнен позднее
    return funcs                  # не сохраняет текущее значение c

for func in odd():
    print(func(), end=' ')        # Опа!: выведет 7 символов g, а не a,b,c,... !
```

Здесь список `func` имитирует зарегистрированные обработчики событий графического интерфейса, подключенные к виджетам. Этот пример работает совсем не так, как могли бы ожидать многие из вас. Переменная `c` внутри вложенной функции здесь всегда будет иметь значение `'g'`, то есть значение, установленное последней итерацией цикла в объемлющей области видимости. В результате все семь сгенерированных `lambda`-функций будут получать при вызове одно и то же значение.

Аналогичная реализация использования дополнительных данных в `lambda`-функциях, вызывающих действительные обработчики графического интерфейса, будет испытывать похожие проблемы. Например, все обработчики событий от кнопок, созданных в цикле, могут в конечном итоге давать один и тот же результат! Чтобы исправить положение, необходимо передавать значения вложенным функциям в виде значений аргументов по умолчанию, которые сохраняют текущее значение переменной цикла (а не то, которое будет в будущем):

```
def odd():
    funcs = []
    for c in 'abcdefg':
        funcs.append((lambda c=c: c)) # запомнить текущее значение c
    return funcs                      # значения по умолчанию вычисляются
                                      # немедленно
for func in odd():
    print(func(), end=' ')            # OK: теперь выведет a,b,c,...
```

Теперь мы получили ожидаемый результат благодаря тому что значения по умолчанию, в отличие от ссылок во внешнюю область видимости, вычисляются в момент *создания* функции, а не в момент ее вызова. При таком подходе сохраняется значение, которое переменная в объемлющей области видимости имела на момент создания функции, а не последнее, присвоенное ей значение. То же правило действует, даже если объемлющей областью видимости для функции является модуль, а не какая-то другая функция – если в следующем фрагменте не использовать аргумент со значением по умолчанию, при обращении к переменной цикла все семь функций получат одно и то же значение:

```
funcs = []                          # объемлющая область видимости - модуль
for c in 'abcdefg':                 # запомнить текущее значение c,
    funcs.append((lambda c=c: c)) # иначе снова выведет 7 символов g

for func in funcs:
    print(func(), end=' ')          # OK: выведет a,b,c,...
```

Из всего вышеизложенного следует вывод, что ссылка на переменную в объемлющей области видимости может служить заменой передачи данных в виде аргумента со значением по умолчанию, *но только* при условии, что эта переменная не получит новое значение, которое вам не нужно, после создания вложенной функции. Вы вообще не должны использовать внутри вложенных функций ссылки на переменные циклов в объемлющей области видимости, потому что они изменяются в процессе выполнения цикла. Однако в большинстве других случаев переменные в объемлющей области видимости могут принимать только одно значение, поэтому ссылки на них можно использовать без опаски.

Мы еще будем сталкиваться с этим явлением в последующих примерах, создающих более сложные графические интерфейсы. А пока запомните, что ссылки в объемлющую область видимости не могут служить полной заменой аргументов со значениями по умолчанию – в некоторых ситуациях значения по умолчанию незаменимы для передачи значений функциям обратного вызова. Кроме того, имейте в виду, что зачастую классы обеспечивают более простой способ сохранения информации для использования в обработчиках, чем вложенные функции. Поскольку информация в классах сохраняется более явным способом, им не свойственны описанные проблемы областей видимости. Следующие два раздела детально рассказывают об использовании классов.

Связанные методы как обработчики событий

Вернемся к теме программирования графических интерфейсов. Во многих случаях вполне можно ограничиться функциями и `lambda`-выражениями, однако связанные методы класса тоже прекрасно справляются с ролью обработчиков событий в графических интерфейсах – они запоминают не только экземпляр, которому было послано событие, но

и связанный с ним метод, который должен быть вызван. Рассмотрим пример 7.14, где приводится переработанная версия примеров 7.12 и 7.13, регистрирующая связанный метод класса вместо функции или результата lambda-выражения.

Пример 7.14. PP4E\Gui\Intro\gui3c.py

```
import sys
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 - связанный метод
        sys.exit()                        # хранит пару self+quit

HelloClass()
mainloop()
```

При нажатии кнопки библиотека tkinter вызовет метод quit этого класса как обычно, без аргументов. Но в действительности он получит один аргумент – оригинальный объект self – хотя tkinter не передает его явно. Поскольку связанный метод self.quit хранит обе ссылки, self и quit, он совместим с вызовом простой функции – интерпретатор Python автоматически передаст аргумент self функции метода. Напротив, регистрация несвязанного метода экземпляра в виде HelloClass.quit работать не станет, потому что в этом случае не будет объекта self, который можно передать потом при возникновении события.

Позднее мы увидим, что приемы использования классов в качестве обработчиков событий также предоставляют естественное место для сохранения данных, используемых при возникновении событий, – достаточно просто присвоить их атрибутам экземпляра self:

```
class someGuiClass:
    def __init__(self):
        self.X = 42
        self.Y = 'spam'
        Button(text='Hi', command=self.handler)
    def handler(self):
        ...использование self.X, self.Y...
```

Поскольку событие будет отправлено методу этого класса со ссылкой на оригинальный объект экземпляра, аргумент self позволит обращаться к атрибутам, содержащим исходные данные. В действительности атрибуты экземпляра сохраняют информацию для использования в процессе обработки события. Этот прием обеспечивает большую гибкость, чем глобальные переменные или дополнительные аргументы, добавляемые

с помощью lambda-выражений, особенно заметную в больших графических интерфейсах.

Объекты вызываемых классов как обработчики событий

Поскольку объекты экземпляров классов в языке Python могут вызываться как функции, если они наследуют метод __call__ для перехвата этой операции, их также можно использовать в качестве обработчиков событий. В примере 7.15 демонстрируется класс, реализующий необходимый интерфейс.

Пример 7.15. PP4E\Gui\Intro\gui3d.py

```
import sys
from tkinter import *

class HelloCallable:
    def __init__(self): # __init__ вызывается при создании объекта
        self.msg = 'Hello __call__ world'

    def __call__(self):
        print(self.msg) # __call__ вызывается при попытке обратиться
        sys.exit()      # к объекту класса как к функции

widget = Button(None, text='Hello event world', command=HelloCallable())
widget.pack()
widget.mainloop()
```

Здесь экземпляр класса HelloCallable, зарегистрированный в command, тоже может вызываться как обычная функция – Python вызовет его метод __call__ для обработки операции вызова, выполняемой в tkinter при нажатии кнопки. В данном случае обобщенный метод __call__ фактически замещает связанный метод. Обратите внимание, что здесь для хранения информации, используемой при обработке событий, задействуется атрибут self.msg – self, являющийся ссылкой на исходный экземпляр класса, который автоматически передается при вызове специального метода __call__.

Все четыре версии gui3 создают одинаковые окна (рис. 7.11), но при нажатии на кнопку выводят в stdout различные сообщения:

```
C:\...\PP4E\Gui\Intro> python gui3.py
Hello, I must be going...

C:\...\PP4E\Gui\Intro> python gui3b.py
Hello lambda world

C:\...\PP4E\Gui\Intro> python gui3c.py
Hello class method world
```

```
C:\...\PP4E\Gui\Intro> python gui3d.py
Hello __call__ world
```

Существуют свои основания для использования каждого из приемов определения обработчиков событий (функция, `lambda`-выражение, метод класса, вызываемый класс), но необходимо перейти к более крупным примерам, чтобы выйти из области чистого теоретизирования.

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 обработчиков в библиотеке tkinter. На самом деле существуют различные способы перехвата событий в сценариях tkinter:

Параметр command кнопки

Как только что было показано, событие нажатия кнопки перехватывается путем передачи вызываемого объекта в параметре `command` виджета. То же относится к другим виджетам, похожим на кнопки, с которыми мы познакомимся в главе 8 (например, переключателям, флажкам и ползункам).

Параметры command меню

В последующих главах обзора библиотеки tkinter будет показано, что параметр `command` используется также для передачи обработчиков событий выбора пунктов меню.

Протоколы полос прокрутки

Виджеты полос прокрутки тоже регистрируют обработчики с помощью параметра `command`, но обладают особым протоколом событий, позволяющим им быть взаимно связанными с виджетом, для прокрутки которого они предназначены (например, окна списков, текстовые окна и холсты): перемещение ползунка на полосе прокрутки автоматически перемещает содержимое виджета и наоборот.

Обобщенные методы `bind` *виджетов*

Имеется возможность использовать более универсальный механизм метода `bind` для регистрации низкоуровневых обработчиков событий, таких как нажатия клавиш, перемещения и щелчки мышью и так далее. В отличие от обработчиков, зарегистрированных с помощью параметра `command`, обработчики, зарегистрированные методом `bind`, получают в качестве аргумента объект события (экземпляр класса `Event` из библиотеки tkinter), который предоставляет контекст события – виджет, к которому относится событие, экранные координаты и так далее.

Протоколы менеджера окон

Кроме того, сценарии могут перехватывать события менеджера окон (например, запрос на закрытие окна) путем внедрения в механизм метода `protocol` менеджера окон, который доступен для оконных объектов верхнего уровня. Например, установив обработчик события `WM_DELETE_WINDOW`, можно перехватить событие от кнопки закрытия окна.

Обработчики планируемых событий

Наконец, в сценарии, использующем библиотеку tkinter, можно также зарегистрировать обработчики, которые должны вызываться в особых случаях, например при срабатывании таймера, поступлении входных данных и холостом состоянии цикла событий. Сценарии могут также приостанавливаться и ожидать событий, связанных с изменением состояния окон и специальных переменных. С этими типами событий мы более подробно познакомимся в конце главы 9.

Связывание событий

Из всех перечисленных протоколов наиболее универсальным, но, вероятно, и наиболее сложным является метод `bind`. Более подробно мы изучим его потом, но чтобы получить первоначальное представление, рассмотрим пример 7.16, который создает тот же графический интерфейс, что и примеры из предыдущего раздела, но для перехвата события нажатия кнопки использует метод `bind`, а не параметр `command`.

Пример 7.16. PP4E\Gui\Intro\gui3e.py

```
import sys
from tkinter import *

def hello(event):
    print('Press twice to exit')       # одиночный щелчок левой кнопкой

def quit(event):                       # двойной щелчок левой кнопкой
    print('Hello, I must be going...') # event дает виджет, координаты и т.д.
    sys.exit()

widget = Button(None, text='Hello event world')
widget.pack()
widget.bind('<Button-1>', hello)       # привязать обработчик щелчка
widget.bind('<Double-1>', quit)        # привязать обработчик двойного щелчка
widget.mainloop()
```

В этой версии параметр `command` для кнопки вообще не определяется. Вместо этого выполняется установка низкоуровневых обработчиков событий одинарных (`<Button-1>`) и 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 8.

При выполнении этого сценария снова создается то же самое окно (рис. 7.11). При одиночном щелчке на кнопке программа выводит сообщение, но не завершается – чтобы завершить программу, нужно щелкнуть на кнопке дважды. Ниже приводятся сообщения, выводимые после двух одиночных щелчков и одного двойного (двойной щелчок сначала запускает обработчик события одиночного щелчка):

```
C:\...\PP4E\Gui\Intro> python gui3e.py
Press twice to exit
Press twice to exit
Press twice to exit
Hello, I must be going...
```

Щелчки мышью на кнопке в этом сценарии перехватываются вручную, поэтому конечный результат примерно такой же – специальные протоколы виджетов, такие как параметр command кнопки, в действительности являются просто интерфейсами более высокого уровня к событиям, которые могут быть перехвачены с помощью метода bind.

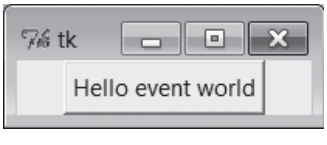


Рис. 7.11. *Кнопка, нажатие на которую вызывает функцию Python*

Более подробно мы разберем работу метода bind и все другие способы установки обработчиков событий, предоставляемые библиотекой tkinter, далее в этой книге. Однако сначала займемся построением более крупных графических интерфейсов, состоящих не только из одной кнопки, и другими способами использования классов в графических интерфейсах.

Добавление нескольких виджетов

Настало время строить интерфейсы пользователя с несколькими виджетами. Сценарий в примере 7.17 создает окно, изображенное на рис. 7.12.

Пример 7.17. PP4E\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()
```

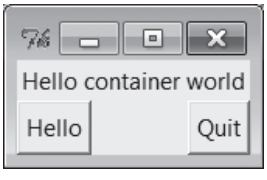


***Рис. 7.12.** Окно с несколькими виджетами*

Этот сценарий создает виджет `Frame` (еще один класс из библиотеки tkinter), к которому прикрепляются три других виджета – `Label` и два `Button` – путем передачи объекта `Frame` в первом аргументе. На языке tkinter это означает, что виджет `Frame` становится родителем для трех других виджетов. Обе кнопки этого интерфейса вызывают следующие обработчики:

- Щелчок на кнопке Hello запускает функцию `greeting`, определенную внутри этого файла, которая производит вывод в поток `stdout`.
- Щелчок на кнопке Quit вызывает стандартный метод tkinter `quit`, который виджет `win` наследует от класса `Frame` (`Frame.quit` имеет тот же эффект, что и использованный ранее метод `Tk.quit`).

Ниже приводится текст, который выводится в `stdout` при щелчке на кнопке Hello, какими бы ни были стандартные потоки ввода-вывода для этого сценария:

```
C:\...\PP4E\Gui\Intro> python gui4.py
Hello stdout world!...
Hello stdout world!...
Hello stdout world!...
Hello stdout world!...
```

Понятие прикрепления виджетов к контейнерам оказывается в сердцевине всех структур в библиотеке tkinter. Однако прежде чем подробно вникать в эту тему, обсудим небольшую деталь.

Еще раз об изменении размеров: обрезание

Ранее мы видели, как заставить виджеты расширяться вместе с родительским окном, передавая параметры `expand` и `fill` менеджеру компоновки `pack`. Теперь, когда у нас в окне имеется несколько виджетов, я открою вам один из полезных секретов компоновщика. Как правило, при уменьшении размеров окна виджеты, добавленные первыми, об-

резаются в последнюю очередь. Это означает, что порядок добавления элементов определяет, какие из них окажутся скрытыми, если окно сделается слишком маленьким, – элементы, добавленные последними, обрезаются в первую очередь. Например, на рис. 7.13 показано, что произойдет, если окно сценария `gui4` уменьшить в интерактивном режиме.

***Рис. 7.13.** Уменьшение размеров gui4*

Попробуйте изменить порядок создания метки и кнопок в сценарии и посмотреть, что произойдет при сжатии окна: виджет, добавленный первым, всегда исчезает последним. Например, если метка добавляется последней, то, как видно на рис. 7.14, она будет обрезана первой, даже если прикреплена к верхнему краю: параметр `side` и порядок добавления виджетов вместе влияют на общую структуру, но только порядок упаковки имеет значение, когда окно уменьшается. Ниже показан измененный порядок создания виджетов:

```
Button(win, text='Hello', command=greeting).pack(side=LEFT)
Button(win, text='Quit',  command=win.quit).pack(side=RIGHT)
Label(win, text='Hello container world').pack(side=TOP)
```

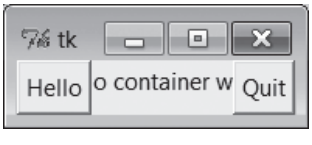


***Рис. 7.14.** Метка добавляется последней, а обрезается первой*

Это обусловлено тем, что библиотека tkinter запоминает порядок добавления. Сценарии могут заранее готовиться к сжатию окон, вызывая вначале метод `pack` для более важных виджетов. Например, в предстоящем обзоре библиотеки tkinter мы встретим сценарий, который создает меню и панели инструментов в верхней и нижней части окна. Чтобы обеспечить их исчезновение при сжатии окна в последнюю очередь, они добавляются первыми, перед теми компонентами, которые размещаются в середине. Аналогично полосы прокрутки, содержащиеся в интерфейсах, обычно добавляются раньше, чем прокручиваемые ими элементы (например, текстовые окна или списки), чтобы сохраняться при сжатии окна.

Прикрепление виджетов к фреймам

Наиболее важным нововведением в этом примере является использование фреймов: виджеты `Frame` служат просто контейнерами для других

виджетов, благодаря чему возникает представление о графических интерфейсах как об иерархиях, или деревьях виджетов. Здесь роль охватывающего окна для трех других элементов играет виджет `win`. Однако в целом, прикрепляя виджеты к фреймам, а фреймы к другим фреймам, можно строить графические интерфейсы с произвольной структурой. Просто поделите графический интерфейс на ряд прямоугольников уменьшающегося размера, реализуйте каждый из них как `Frame` и прикрепите к фреймам простые виджеты.

В данном сценарии, когда виджет `win` передается в первом аргументе конструкторам `Label` и `Button`, библиотека tkinter прикрепляет их к виджету `Frame` (они становятся дочерними для `win`). Сам объект `win` по умолчанию прикрепляется к окну верхнего уровня, потому что конструктору `Frame` не был передан родитель. Когда мы предлагаем виджету `win` начать выполнение (вызывая метод `mainloop`), библиотека tkinter отображает все графические элементы в построенном нами дереве.

Три дочерних виджета также позволяют указывать параметры `pack`: аргументы `side` говорят о том, к какой части содержащего их фрейма (то есть `win`) должен быть прикреплен новый виджет. 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 виджетов происходит немного сложнее, чем простое указание сторон, к которым они прикрепляются, но чтобы узнать почему, придется сделать краткое отступление и обсудить детали работы менеджера компоновки.

Порядок компоновки и прикрепление к сторонам

При отображении дерева виджетов дочерние виджеты появляются внутри родительских и располагаются в соответствии с порядком и параметрами компоновки. По этой причине порядок добавления элементов не только определяет порядок их обрезания, но также определяет, как будут учитываться значения параметра `side` в общей картине.

Вот как работает система компоновки элементов:

1. Компоновщик начинает с пустого доступного пространства, в которое входит весь родительский контейнер (например, весь фрейм или окно верхнего уровня).
2. Когда виджет прикрепляется к какому-либо краю, ему отдается весь запрашиваемый край в оставшемся пустом пространстве, и пустое пространство сокращается.
3. Последующие виджеты получают все, что осталось от этого края после добавления предыдущих виджетов.
4. После того как виджетам будет отдано все пустое пространство, `expand` делит оставшееся пространство, а `fill` и `anchor` растягивают и устанавливают виджеты внутри выделенной им области.

Например, изменим в сценарии gui4 логику создания дочерних виджетов, как показано ниже:

```
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, изображенный на рис. 7.15, и это всего лишь из-за перемещения инструкции создания метки на одну строку вниз (сравните с рис. 7.12).

Рис. 7.15. *Метка была добавлена второй*

Теперь, несмотря на значение параметра side, метка не захватывает весь верх окна, и чтобы разобраться в причине, нужно представить себе *сокращающиеся пустые пространства*. Так как первой в интерфейс добавляется кнопка Hello, ей выделяется весь левый край фрейма. 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 добавлению: первой исчезает кнопка Quit, за ней следует метка.[1]

В первоначальной версии этого примера (рис. 7.12) метка занимает весь верхний край только потому, что она добавляется первой, а не благодаря значению параметра side. Если внимательнее рассмотреть рис. 7.14, можно заметить, что он иллюстрирует действие тех же самых правил – метка находится между кнопками, потому что они уже полностью заняли левый и правый край.

Снова о параметрах expand и fill компоновки

Помимо порядка добавления на взаимное расположение виджетов оказывает влияние также уже знакомый параметр fill, позволяющий растягивать виджет так, чтобы он занимал все пространство вдоль выделенного ему пустого края, а все пустое пространство, оставшееся после расстановки элементов, поровну распределять между виджетами, при

[1] Технически после изменения размеров окна этапы добавления просто выполняются заново. Но поскольку это означает, что для графических элементов, добавляемых последними, останется недостаточно места, это равносильно тому, что элементы, добавленные последними, обрезаются первыми.

добавлении которых был указан параметр expand=YES. Например, следующий фрагмент создает окно, изображенное на рис. 7.16 (сравните с рис. 7.15).

```
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)
```

Рис. 7.16. *Компоновка с параметрами expand и fill*

Чтобы все эти элементы увеличивались вместе с окном, необходимо сделать содержащий их контейнер расширяемым – виджеты расширяются за пределы своего первоначального положения, только если *расширяются все их родители*. Ниже демонстрируются изменения в *gui4.py*:

```
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 фрагмента фрейм получит весь верхний край родительского окна, как и раньше (то есть верхний участок корневого окна). Однако теперь он настроен так, что должен расширяться и заполнять неиспользуемое пространство своего родителя в обоих направлениях, поэтому он и все прикрепленные к нему дочерние элементы расширяются вместе с окном, как показано на рис. 7.17.

Рис. 7.17. *Расширяемый фрейм в увеличенном окне gui4*

Использование якорей вместо растягивания

Как если бы это не обеспечивало достаточной гибкости, механизм компоновки дополнительно предоставляет для размещения графических элементов в отведенной для них области параметр `anchor` помимо заполнения пространства с помощью `fill`. Параметр `anchor` принимает константы из библиотеки tkinter, указывающие восемь направлений (`N`, `NE`, `NW`, `S` и так далее), или константу `CENTER` (например, `anchor=NW`). Компоновщику при этом предписывается разместить виджет в желательном месте внутри отведенного для него пространства, если это пространство больше, чем требуется для изображения данного графического элемента.

По умолчанию параметр `anchor` получает значение `CENTER`, поэтому виджеты выводятся в центре отведенного им пространства (выделенного им края пустого пространства), если только для них не определено иное местоположение с помощью параметра `anchor` или они не растянуты с помощью параметра `fill`. Для демонстрации изменим сценарий `gui4`, как показано ниже:

```
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 добавлена первой, она получила весь левый край родительского фрейма. Места оказалось больше, чем необходимо кнопке, поэтому по умолчанию она оказывается в середине этого края, как показано на рис. 7.15 (то есть закреплена в центре). После установки якоря в значение `N` она смещается вверх по левому краю, как показано на рис. 7.18.

***Рис. 7.18.** Закрепление кнопки на севере*

Имейте в виду, что параметры `fill` и `anchor` начинают приниматься во внимание только после того как виджету будет выделено место у края пустого пространства, определяемого параметром `side`, в соответствии с порядком добавления, и запроса дополнительного пространства `expand`. Путем варьирования порядка добавления, а также значений параметров, определяющих край, направление заполнения и закрепления, можно получить массу эффектов расположения и обрезания, и стоит потратить некоторое время, экспериментируя с разными вариантами, если вы этого еще не сделали. Например, в исходной версии этого при-

мера метка занимает весь верхний край только потому, что была добавлена первой.

Как будет показано ниже, фреймы можно вкладывать в другие фреймы, благодаря чему можно создавать довольно сложные структуры. В действительности, каждый родительский контейнер является отдельным пустым пространством, поэтому возникает некоторый механизм обхода для алгоритма выделения пустого пространства компоновщиком: чтобы точнее управлять местом, где будет отображаться некоторая группа виджетов, просто скомпонуйте их во вложенном фрейме и прикрепите этот фрейм как пакет к более крупному контейнеру. Например, ряд расположенных рядом кнопок проще поместить в отдельный фрейм, чем смешивать с остальными виджетами.

Наконец, имейте в виду, что дерево виджетов, создаваемое в этих примерах, в действительности является неявным – библиотека tkinter внутри ведет учет связей, которые образуются в результате передачи аргументов родительских виджетов. На языке ООП это называется *композицией* – фрейм содержит метку и несколько кнопок. А теперь рассмотрим другой вид связи – *наследование*.

Настройка виджетов с помощью классов

В сценариях, использующих библиотеку tkinter, совсем необязательно применять приемы ООП, но они определенно могут оказаться полезными. Как мы только что видели, графические интерфейсы на базе tkinter строятся, как деревья объектов экземпляров классов. Ниже демонстрируется еще один способ применения ООП для моделирования графического интерфейса: специализация виджетов посредством наследования. Сценарий в примере 7.18 создает окно, изображенное на рис. 7.19.

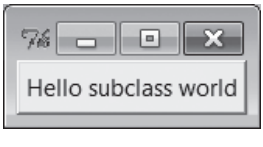


Рис. 7.19. *Подкласс кнопки в действии*

Пример 7.18. PP4E\Gui\Intro\gui5.py

```
from tkinter import *

class HelloButton(Button):
    def __init__(self, parent=None, **config):  # регистрирует метод callback
        Button.__init__(self, parent, **config) # и добавляет себя в интерфейс
        self.pack()                             # можно использовать старый
        self.config(command=self.callback)      # стиль аргумента config

    def callback(self):                     # действие по умолчанию при нажатии
```

```
        print('Goodbye world...')          # переопределить в подклассах
        self.quit()

if __name__ == '__main__':
    HelloButton(text='Hello subclass world').mainloop()
```

В этом примере нет ничего необычного: он просто отображает одну кнопку, при нажатии на которую программа выводит сообщение и завершается. Но на этот раз мы сами создали виджет кнопки. Класс `HelloButton` наследует все свойства и методы класса `Button`, а также добавляет метод `callback` и логику в конструкторе, устанавливая параметру `command` значение `self.callback` – связанный метод экземпляра. При нажатии на кнопку теперь вызывается не просто функция, а метод `callback` нового класса виджета.

Здесь аргумент `**config` собирает в словарь все дополнительные именованные аргументы, которые затем передаются конструктору `Button`. Конструкция `**config` в вызове конструктора `Button` распаковывает словарь в список именованных аргументов (в действительности в этом нет необходимости, благодаря поддержке устаревшей формы вызова со словарем, встречавшейся нам ранее, но и вреда никакого не будет). Мы уже встречались с вызовом метода `config` виджетов в конструкторе `HelloButton`: это просто альтернативный способ передачи параметров настройки после создания виджета (вместо передачи аргументов конструктору).

Стандартизация поведения и внешнего вида

Какой же смысл в создании подклассов виджетов? В двух словах, благодаря такому подходу появляется возможность создавать виджеты, выглядящие и действующие одинаково, путем создания подклассов, и использовать преимущества модели ООП в языке Python. Этот прием может оказаться достаточно мощным, чтобы использовать его в крупных программах.

Общие черты поведения

Сценарий в примере 7.18 стандартизирует поведение – он демонстрирует возможность настройки виджетов путем создания подклассов вместо передачи параметров. Экземпляр класса `HelloButton` является настоящей кнопкой – при ее создании параметры настройки, такие как `text`, передаются как обычно. Но можно также определить обработчик событий, переопределив в подклассе метод `callback`, как показано в примере 7.19.

Пример 7.19. PP4E\Gui\Intro\gui5b.py

```
from gui5 import HelloButton

class MyButton(HelloButton):               # подкласс класса HelloButton
    def callback(self):                    # переопределяет метод обработчика
        print("Ignoring press!...")        # события нажатия кнопки
```

```
if __name__ == '__main__':
    MyButton(None, text='Hello subclass world').mainloop()
```

Этот сценарий создает то же самое окно, но вместо завершения программы при нажатии кнопки `MyButton` происходит вывод сообщения в поток `stdout` и программа продолжает работу. Ниже показаны сообщения, выведенные в стандартный поток вывода после нескольких нажатий:

```
C:\...\PP4E\Gui\Intro> python gui5b.py
Ignoring press!...
Ignoring press!...
Ignoring press!...
Ignoring press!...
```

Решить, как проще настраивать графические элементы – создавая подклассы или путем передачи им параметров, – это дело вкуса. Но нужно отметить, что библиотека Tk приобретает в Python настоящие объектно-ориентированные черты, поскольку объектно-ориентированным является сам язык Python – используя привычные приемы ООП, мы можем создавать свои подклассы виджетов. Такой подход позволяет определять не только общие черты поведения, но и общий внешний вид.

Общий внешний вид

Например, хотя мы и не будем углубляться в изучение параметров настройки виджетов до следующей главы, тем не менее подобный класс кнопки мог бы обеспечить стандартизацию внешнего вида, отличного от внешнего вида, придаваемого библиотекой tkinter по умолчанию, для каждого своего экземпляра, и приблизиться к реализации таких понятий, как «стили» или «темы», используемых в некоторых других инструментах создания графических интерфейсов:

```
class ThemedButton(Button):                       # настраивает стиль
    def __init__(self, parent=None, **configs):  # для всех экземпляров
        Button.__init__(self, parent, **configs) # описание параметров
        self.pack()                               # смотрите в главе 8
        self.config(fg='red', bg='black',
                    font=('courier', 12), relief=RAISED, bd=5)

B1 = ThemedButton(text='spam', command=onSpam)   # обычные виджеты кнопок
B2 = ThemedButton(text='eggs')                    # но наследуют общий стиль
B2.pack(expand=YES, fill=BOTH)
```

Этот фрагмент приводится лишь для предварительного ознакомления – полную версию вы найдете в файле *gui5b-themed.py* в дереве примеров, а дополнительные сведения о параметрах настройки виджетов – в главе 8. Но он иллюстрирует возможность придания общего внешнего вида за счет создания подклассов виджетов – все кнопки, созданные из этого класса, будут выглядеть одинаково, и автоматически будут изменять внешний вид при любых изменениях в классе.

Прием создания подклассов – это, безусловно, инструмент программиста, но мы можем сделать возможность настройки доступной и для пользователей графических интерфейсов. В крупных программах, демонстрируемых далее в этой книге (например, PyEdit, PyClock и PyMailGUI), мы иногда будем добиваться похожего эффекта за счет импортирования настроек из модулей и применения их к виджетам, как если бы они были встроенными настройками. Если такие внешние настройки использовать в подклассах виджетов, таких как наш класс `ThemedButton` выше, они будут применяться ко всем экземплярам и подклассам (для справки: полная версия следующего фрагмента находится в файле *gui5b-themed-user.py*):

```
from user_preferences import bcolor, bfont, bsize # получить настройки

class ThemedButton(Button):
    def __init__(self, parent=None, **configs):
        Button.__init__(self, parent, **configs)
        self.pack()
        self.config(bg=bcolor, font=(bfont, bsize))

ThemedButton(text='spam', command=onSpam) # обычные виджеты кнопок, но
ThemedButton(text='eggs', command=onEggs) # наследуют настройки пользователя

class MyButton(ThemedButton):                    # подклассы также наследуют
    def __init__(self, parent=None, **configs): # настройки пользователя
        ThemedButton.__init__(self, parent, **configs)
        self.config(text='subclass')

MyButton(command=onSpam)
```

Напомню, что подробнее о параметрах настройки виджетов будет рассказываться в следующей главе, а здесь я хотел лишь донести общую мысль, что настройка виджетов путем создания подклассов позволяет определять не только поведение, но и внешний вид всего набора виджетов. Следующий пример демонстрирует еще один способ настройки – создание настраиваемых и присоединяемых пакетов виджетов, обычно известных, как *компоненты*.

Повторно используемые компоненты и классы

Крупные графические интерфейсы часто строятся как подклассы `Frame` с реализацией обработчиков событий в виде методов. Благодаря такой структуре возникает естественное место для хранения информации между событиями: атрибуты экземпляров классов. Она позволяет также специализировать графические интерфейсы за счет переопределения методов в новых подклассах и прикреплять их к более крупным структурам интерфейса, чтобы повторно использовать в качестве общих компонентов. Например, текстовый редактор с графическим интерфейсом, реализованный как подкласс `Frame`, можно прикреплять

к любому числу других графических интерфейсов и настраивать с их помощью – при правильном использовании такой текстовый редактор можно встроить в любой интерфейс пользователя, где требуются средства редактирования текста.

С таким текстовым редактором в виде компонента мы встретимся в главе 11. А пока проиллюстрируем идею простым примером 7.20. Сценарий *gui6.py* создает окно, изображенное на рис. 7.20.

Пример 7.20. PP4E\Gui\Intro\gui6.py

```
from tkinter import *

class Hello(Frame):                    # расширенная версия класса Frame
    def __init__(self, parent=None):
        Frame.__init__(self, parent)   # вызвать метод __init__ суперкласса
        self.pack()
        self.data = 42
        self.make_widgets()            # прикрепить виджеты к себе

    def make_widgets(self):
        widget = Button(self, text='Hello frame world!', command=self.message)
        widget.pack(side=LEFT)

    def message(self):
        self.data += 1
        print('Hello frame world %s!' % self.data)

if __name__ == '__main__': Hello().mainloop()
```

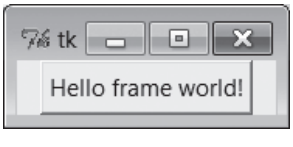


***Рис. 7.20.** Нестандартный фрейм в действии*

Этот сценарий выводит окно с единственной кнопкой. При ее нажатии вызывается связанный метод `self.message`, который снова выводит сообщение в `stdout`. Ниже показаны сообщения, выведенные после четырехкратного нажатия кнопки – обратите внимание, что атрибут `self.data` (в данном случае простой счетчик) сохраняет информацию о состоянии между нажатиями:

```
C:\...\PP4E\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 отображения кнопки типа `Button` (в примерах 7.10, 7.11 и 7.12 то же достигалось меньшим числом

строк). Но класс `Hello` предоставляет контейнерную организационную *структуру* для построения графического интерфейса. В примерах, предшествовавших предыдущему разделу, графические интерфейсы создавались с применением процедурного подхода: мы вызывали конструкторы виджетов, как если бы они были функциями, и связывали виджеты воедино, указывая родителя при вызове конструктора. Не было никакого представления о внешнем контексте, помимо глобальной области видимости файла модуля, содержащего вызовы методов виджетов. Такой подход годится для простых графических интерфейсов, но при построении структур более крупных интерфейсов служит причиной хрупкости кода.

При создании подкласса, наследующего класс `Frame`, как было продемонстрировано, этот класс становится охватывающим контекстом графического интерфейса:

- Виджеты добавляются путем прикрепления объектов к `self`, экземпляру подкласса контейнера `Frame` (например, `Button`).
- Обработчики событий регистрируются как связанные методы объекта `self`, вследствие чего вызовы направляются обратно в реализацию класса (например, `self.message`).
- Информация о состоянии сохраняется между событиями путем присвоения атрибутам объекта `self` и доступна всем обработчикам событий в классе (например, `self.data`).
- Легко можно создать несколько экземпляров такого компонента графического интерфейса, даже внутри одного и того же процесса, потому что каждый экземпляр класса является отдельным пространством имен.
- Классы обладают естественной возможностью настройки благодаря возможности наследования и композиции.

В некотором смысле графические интерфейсы целиком становятся специализированными объектами `Frame` с расширениями, соответствующими их применению. Классы также позволяют реализовать протоколы построения виджетов (например, метод `make_widgets` в данном случае), выполнять стандартные задачи настройки (например, установка параметров настройки менеджера окон) и так далее. Одним словом, подклассы `Frame` дают простой способ организации совокупностей объектов других классов виджетов.

Прикрепление классов компонентов

Более важно, вероятно, что подклассы `Frame` являются настоящими виджетами: их можно и дальше расширять и модифицировать, создавая подклассы, и прикреплять к контейнерным виджетам. Например, чтобы прикрепить весь пакет виджетов, создаваемых классом, к чему-либо еще, нужно создать экземпляр класса, передав ему родительский

виджет. Это иллюстрируется примером 7.21, при запуске которого создается окно, изображенное на рис. 7.21.

Пример 7.21. PP4E\Gui\Intro\gui6b.py

```
from sys import exit
from tkinter import *          # импортировать классы виджетов Tk
from gui6 import Hello         # импортировать подкласс фрейма

parent = Frame(None)           # создать контейнерный виджет
parent.pack()
Hello(parent).pack(side=RIGHT) # прикрепить виджет Hello, не запуская его

Button(parent, text='Attach', command=exit).pack(side=LEFT)
parent.mainloop()
```

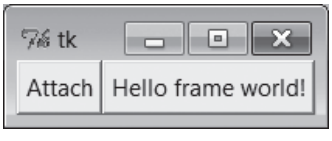


***Рис. 7.21.** Прикрепленный компонент класса справа*

В этом сценарии кнопка `Hello` добавляется к правому краю родителя `parent` – контейнера `Frame`. На самом деле, кнопка в правой части этого окна является встроенным компонентом: его кнопка действительно представляет прикрепленный объект класса Python. Нажатие кнопки встроенного класса справа, как и ранее, выводит сообщение; нажатие новой кнопки закрывает окно вызовом `sys.exit`:

```
C:\...\PP4E\Gui\Intro> python gui6b.py
Hello frame world 43!
Hello frame world 44!
Hello frame world 45!
Hello frame world 46!
```

В более сложных интерфейсах можно прикреплять большие подклассы `Frame` к другим компонентам-контейнерам и разрабатывать каждый из них независимо. В примере 7.22 демонстрируется еще один специализированный подкласс класса `Frame`, который прикрепляет экземпляр класса `Hello` более объектно-ориентированным способом. При запуске в виде самостоятельного сценария он создает окно, идентичное изображенному на рис. 7.21.

Пример 7.22. PP4E\Gui\Intro\gui6c.py

```
from tkinter import *          # импортировать классы виджетов Tk
from gui6 import Hello         # импортировать подкласс фрейма

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 объект класса Hello к себе
        Button(self, text='Attach', command=self.quit).pack(side=LEFT)

if __name__ == '__main__': HelloContainer().mainloop()
```

Выглядит и действует этот сценарий в точности как `gui6b`, но в качестве обработчика события добавленной кнопки он регистрирует метод `self.quit`, который является стандартным методом `quit` виджетов, унаследованным от `Frame`. На этот раз окно демонстрирует действие двух классов Python – виджеты встроенного компонента справа (оригинальная 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 виджетов может быть значительно больше. Представьте себе, что мы заменили вызов конструктора `Hello` в этом сценарии вызовом, прикрепляющим готовый и полностью отлаженный объект калькулятора, и мощь данной парадигмы станет вам более понятна. Если все свои компоненты графического интерфейса создавать как классы, они автоматически образуют библиотеку многократно используемых виджетов, которые всегда можно будет вставить в другие приложения.

Расширение классов компонентов

При построении графических интерфейсов с помощью классов существует ряд способов повторного их использования в других приложениях. Чтобы расширить класс `Hello`, а не прикреплять его, можно переопределить некоторые его методы в новом подклассе (который сам станет специализированным виджетом `Frame`). Этот прием демонстрируется в примере 7.23.

Пример 7.23. PP4E\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 создает виджеты обращением к методу суперкласса, а затем добавляет справа вторую кнопку Extend, как показано на рис. 7.22.

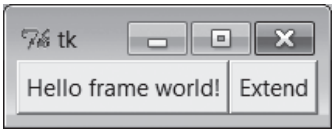


***Рис. 7.22.** Модифицированные виджеты класса слева*

Поскольку этот подкласс переопределяет метод message, нажатие кнопки исходного суперкласса, расположенной слева, теперь выводит в stdout другую строку (когда осуществляется поиск вверх по дереву наследования, начиная от объекта self, первым обнаруживается атрибут message в этом подклассе, а не в суперклассе):

```
C:\...\PP4E\Gui\Intro> python gui6d.py
hello 42
hello 42
hello 42
hello 42
```

Но нажатие новой кнопки Extend справа, добавленной этим подклассом, приводит к немедленному выходу из приложения, потому что обработчиком событий от добавленной кнопки является метод quit (унаследованный от Hello, который в свою очередь наследует его от Frame). Таким образом, этот класс модифицирует исходный класс, добавляя новую кнопку и изменяя поведение метода message.

Это довольно простой пример, но он наглядно демонстрирует технологию, обеспечивающую широкие возможности на практике: чтобы изменить поведение графического интерфейса, можно не изменять существующую реализацию, а написать новый класс, который модифицирует некоторые части интерфейса. Основная реализация отлаживается только один раз, а затем модифицируется с помощью подклассов, когда возникают особые потребности.

Мораль заключается в том, что графические интерфейсы, использующие библиотеку tkinter, можно программировать вообще без создания новых классов, но использование классов для структурирования программного кода в конечном итоге значительно расширяет возможности его повторного использования. При правильном подходе можно не только прикреплять уже отлаженные компоненты к новым интерфейсам, но и изменять их поведение в новых внешних подклассах в соответствии с особыми требованиями. В любом случае первоначальные затраты, связанные с использованием классов, должны в итоге окупиться за счет сокращения времени разработки.

Автономные классы-контейнеры

Прежде чем двинуться дальше, хочу отметить, что большая часть перечисленных выше преимуществ от создания компонентов на основе классов может быть получена в результате создания автономных классов, не являющихся производными от класса `Frame` или других классов виджетов из библиотеки tkinter. Так, класс из примера 7.24 создает окно, изображенное на рис. 7.23.

Пример 7.24. PP4E\Gui\Intro\gui7.py

```
from tkinter import *

class HelloPackage:                     # не является подклассом виджета
    def __init__(self, parent=None):
        self.top = Frame(parent)        # встроить фрейм Frame
        self.top.pack()
        self.data = 0
        self.make_widgets()             # прикрепить виджеты 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 += 1
        print('Hello number', self.data)

if __name__ == '__main__': HelloPackage().top.mainloop()
```

***Рис. 7.23.** Автономный класс в действии*

Если запустить этот сценарий, кнопка `Hye` будет производить вывод в `stdout`, а `Bye` – закрывать окно и завершать работу программы, как и раньше:

```
C:\...\PP4E\Gui\Intro> python gui7.py
Hello number 1
Hello number 2
Hello number 3
Hello number 4
```

Так же как и раньше, атрибут `self.data` сохраняет информацию о состоянии между событиями, а для обработки событий вызывается метод `self.message`, имеющийся в этом классе. Но в отличие от того, что было раньше, сам класс `HelloPackage` не является подклассом виджета `Frame`.

Он вообще не является подклассом какого-либо виджета – он служит только для создания пространства имен, хранящего действительные объекты виджетов и информацию о состоянии. По этой причине виджеты прикрепляются к объекту `self.top` (встроенному фрейму `Frame`), а не к `self`. Более того, все ссылки на объект как на виджет должны передаваться вниз, встроенному фрейму, как, например, вызов метода `top.mainloop` для запуска интерфейса в конце сценария.

Это приводит к тому, что внутри класса приходится писать чуть больше программного кода, но устраняет возможные конфликты имен между атрибутами, добавляемыми к `self` в структуре tkinter и существующими методами виджетов tkinter. Например, если определить в своем классе метод `config`, он замаскирует вызов `config`, экспортируемый библиотекой tkinter. Для автономных классов, как в этом примере, доступны будут только те методы и атрибуты экземпляров, которые определены в этих классах.

В действительности в библиотеке tkinter используется не очень много имен, поэтому обычно это не создает больших проблем.[1] Конечно, такая вероятность существует, но, честно говоря, за 18 лет программирования на языке Python я не встречался с конфликтами имен tkinter в подклассах виджетов. Кроме того, использование автономных классов не лишено своих недостатков. Хотя в целом их можно прикреплять и создавать производные от них классы, они не вполне совместимы с действительными объектами виджетов. Например, вызовы настройки, выполняемые в примере 7.21 для подкласса `Frame`, будут терпеть неудачу в примере 7.25.

Пример 7.25. PP4E\Gui\Intro\gui7b.py

```
from tkinter import *
from gui7 import HelloPackage # или from gui7c, где добавлен __getattr__
```

[1] Если заглянуть в исходные тексты модуля `tkinter` (главным образом в файл `__init__.py` в каталоге `Lib\tkinter`), 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 за пределами реализации tkinter (например, `self.master`, `self.children`). Странно, но библиотека tkinter по-прежнему не использует трюк с «псевдочастными» атрибутами Python, имена которых начинаются с двух символов подчеркивания, что влечет за собой автоматическое добавление имени содержащего их класса и локализации для создающего класса. Если библиотека tkinter будет когда-нибудь переписана так, чтобы использовать эту особенность, конфликты имен в подклассах виджетов будут происходить гораздо реже. Большинство атрибутов классов виджетов являются методами, предназначенными для использования в клиентских сценариях; имена, начинающиеся с одного символа подчеркивания, могут использоваться тоже, но при этом вероятность конфликтов имен все-таки ниже.

```
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()
```

Этот сценарий вообще не будет работать, потому что `part` не является настоящим виджетом. Чтобы работать с ним как с виджетом, нужно спуститься в `part.top`, прежде чем настраивать интерфейс, и рассчитывать на то, что имя `top` не будет изменено разработчиком класса. Другими словами, требуется знать внутреннее устройство класса. Лучше всего эти действия реализовать в самом классе, определив метод, всегда направляющий обращения к неизвестным атрибутам встроенному объекту класса `Frame`, как показано в примере 7.26.

Пример 7.26. PP4E\Gui\Intro\gui7c.py

```
import gui7
from tkinter import *

class HelloPackage(gui7.HelloPackage):
    def __getattr__(self, name):
        return getattr(self.top, name) # передать вызов настоящему виджету

if __name__ == '__main__': HelloPackage().mainloop() # вызовет __getattr__!
```

Этот сценарий создает такое же окно, как на рис. 7.23. Однако изменения в примере 7.25, выражающиеся в импортировании расширенной версии класса `HelloPackage` из модуля `gui7c`, обеспечивают корректную работу интерфейса, изображенного на рис. 7.24.

***Рис. 7.24.** Автономный класс в действии*

Перенаправление операций обращения к атрибутам вложенных виджетов обеспечивает нормальную работу этого примера, но при этом требуется писать еще больше программного кода в автономных классах. Впрочем, как обычно, значимость всех таких компромиссов зависит от конкретного приложения.

Завершение начального обучения

В данной главе мы изучили основы использования библиотеки tkinter в сценариях на языке Python и познакомились с рядом простых виджетов – метками, кнопками, фреймами, а также с менеджером компоновки. Этого достаточно для создания простых интерфейсов, но в действительности мы лишь поверхностно ознакомились с набором виджетов tkinter.

В следующих двух главах мы будем применять полученные знания для изучения оставшейся части библиотеки tkinter и приемов ее использования для создания интерфейсов, встречающихся в реальных программах с графическим интерфейсом. В качестве предварительного обзора и предлагаемого маршрута в табл. 7.1 перечислены виджеты, которые нам там встретятся, примерно в том порядке, в котором они будут появляться. Обратите внимание, что в этой таблице перечислены только классы виджетов. Мы также встретимся с некоторыми дополнительными темами, относящимися к виджетам, которых нет в этой таблице.

Таблица 7.1. Классы виджетов tkinter

Классы виджетов	Описание
`Label`	Простая область для вывода текста
`Button`	Простая кнопка с меткой
`Frame`	Контейнер для прикрепления и размещения других виджетов
`Toplevel, Tk`	Новое окно, управляемое менеджером окон
`Message`	Метка с несколькими строками
`Entry`	Простое однострочное поле ввода текста
`Checkbutton`	Кнопка с двумя состояниями; обычно используется для организации для выбора нескольких вариантов
`Radiobutton`	Кнопка с двумя состояниями; обычно используется для организации выбора одного варианта из нескольких
`Scale`	Ползунок со шкалой
`PhotoImage`	Объект графического изображения, используемый для вывода полноцветных изображений на других виджетах
`BitmapImage`	Объект графического изображения, используемый для вывода растровых изображений на других виджетах
`Menu`	Набор вариантов выбора, связанных с `Menubutton` или с окном верхнего уровня

Таблица 7.1 (продолжение)

Классы виджетов	Описание
`Menubutton`	Кнопка, открывающая меню или подменю с вариантами выбора
`Scrollbar`	Элемент управления для прокрутки содержимого других виджетов (например, списка, холста, текста)
`Listbox`	Список имен, доступных для выбора
`Text`	Виджет для просмотра/редактирования многострочного текста, поддерживающий шрифты и так далее
`Canvas`	Область для изображения графики с поддержкой возможности рисования линий, окружностей, фотографий, текста и так далее.

В этой главе мы уже встречались с виджетами `Label`, `Button` и `Frame`. Для облегчения усвоения оставшийся материал разбит на две главы: глава 8 освещает элементы в верхней части табл. 7.1, вплоть до `Menu`, а в главе 9 представлены виджеты, находящиеся в нижней части таблицы.

Помимо классов виджетов, представленных в табл. 7.1, в библиотеке tkinter содержатся дополнительные классы и инструменты, многие из которых также будут исследованы в двух следующих главах:

Управление компоновкой

`pack`, `grid`, `place`

Связанные переменные tkinter

`StringVar`, `IntVar`, `DoubleVar`, `BooleanVar`

Улучшенные виджеты Tk

`Spinbox`, `LabelFrame`, `PanedWindow`

Составные виджеты

`Dialog`, `ScrolledText`, `OptionMenu`

Планируемые обратные вызовы

Методы виджетов `after`, `wait` и `update`

Прочие инструменты

Стандартные диалоги, буфер обмена, `bind` и `Event`, параметры настройки виджетов, пользовательские и модальные диалоги, анимационные эффекты

Большинство виджетов tkinter принадлежит к числу знакомых интерфейсных элементов. Некоторые из них обладают очень богатой функциональностью. Например, класс `Text` реализует сложный виджет многострочного текста, поддерживающий шрифты, цвета и спецэффекты,

мощности которого достаточно для реализации веб-броузера. Аналогично класс `Canvas` предоставляет множество инструментов, достаточно мощных для создания приложений отображения и обработки изображений. Кроме того, расширения для библиотеки tkinter, такие как Pmw, Tix и ttk, добавляют в инструментарий разработчика графических интерфейсов виджеты с еще более богатыми возможностями.

Соответствие между Python/tkinter и Tcl/Tk

В начале этой главы я упомянул, что библиотека tkinter является интерфейсом Python к библиотеке 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. Кроме того, такое сопоставление сделает для разработчиков программ на языке Python более полезными справочники по Tk, написанные для других языков.

Вообще говоря, ориентированность Tcl на применение строковых команд сильно отличается от подхода к программированию на языке Python, основанного на объектах. Однако, что касается использования Tk, синтаксические различия невелики. Ниже приводятся некоторые главные отличия интерфейса Python к tkinter:

Создание

Виджеты создаются как экземпляры классов при вызове конструктора класса виджета.

Владельцы (родители)

Родителями являются ранее созданные объекты, передаваемые конструкторам классов виджетов.

Параметры виджетов

Параметры являются именованными аргументами конструктора или метода `config` либо ключами словарей.

Операции

Операции виджетов (действия) становятся методами объектов классов виджетов tkinter.

Обратные вызовы

Обработчиком события может быть любой вызываемый объект: функция, метод, `lambda`-выражение и так далее.

Расширение

Виджеты расширяются с использованием механизма наследования классов в языке Python.

Композиция

Интерфейсы конструируются путем прикрепления объектов, а не конкатенации имен.

Связанные переменные (следующая глава)

Переменные, ассоциируемые с виджетами, являются объектами классов tkinter с методами.

Команды создания виджетов в языке Python (например, `button`) являются именами классов, начинающимися с заглавной буквы (например, `Button`), операции с виджетами, состоящие 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 8 будет также показано, что «переменные» библиотеки tkinter, ассоциируемые с виджетами, принимают форму объектов экземпляров классов (например, `StringVar`, `IntVar`) с методами `get` и `set`, а не просто именами переменных Python или Tcl. В табл. 7.2 более конкретно приведены основные соответствия между языками.

Таблица 7.2. Соответствие между Tk и tkinter

Операция	Tcl/Tl	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()`
Компоновка	`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, определяющих родителя и передаваемых конструктору, не требуя объединения строк путей к виджетам. После создания виджета к нему можно обращаться непосредственно, по ссылке. При программировании на языке Tcl иногда можно спрятать записанные через точку имена путей вручную в переменных, но это не то же самое, что чисто объектно-ориентированная модель Python.

Как только вы напишете на языке Python несколько сценариев, использующих библиотеку tkinter, отличия объектов Python наверняка будут казаться вам тривиальными. В то же время, поддержка объектно-ориентированных приемов в языке Python вносит совершенно новую составляющую в разработку для Tk: вы получаете те же самые виджеты плюс поддержку возможности структурирования и многократного использования программного кода.

8

Экскурсия по tkinter, часть 1

«Виджеты, гаджеты, графические интерфейсы... Бог мой!»

В этой главе будет продолжено рассмотрение приемов программирования графических интерфейсов на языке Python. В предыдущей главе рассматривались простые виджеты – кнопки, метки и другие – демонстрирующие основы использования библиотеки tkinter в сценариях на языке Python. Такое упрощение было намеренным: легче охватить взглядом картину графического интерфейса целиком, если не приходится вникать в детали интерфейса виджетов. Но теперь, после знакомства с основами, в этой и следующей главе мы переходим к представлению более сложных объектов виджетов и средств, предоставляемых библиотекой tkinter.

Вы увидите, как разработка сценариев с графическим интерфейсом станет полезным и интересным делом. 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 так далее. За этими главами последует еще одна, завершающая, посвященная графическим интерфейсам и рассматривающая еще более крупные примеры, в которых применяются приемы и интерфейсы, демонстрировавшиеся во всех предшествующих главах, в которых говорилось о создании графических интерфейсов. В этих же двух главах примеры будут небольшими и самодостаточными, чтобы позволить сосредоточиться на особенностях виджетов.

Темы этой главы

Формально мы уже использовали ряд простых виджетов в главе 7. Пока мы познакомились с классами `Label`, `Button`, `Frame` и `Tk` и попутно изучили понятия управления компоновкой в методе `pack`. Несмотря на свою простоту, все эти классы достаточно полно представляют интерфейсы библиотеки tkinter в целом и служат рабочими лошадками в типичных графических интерфейсах. Например, контейнеры `Frame` служат основой иерархической структуры отображения.

В этой и следующей главах мы исследуем дополнительные параметры уже знакомых графических элементов и, отойдя от основ, расскажем об остальной части набора виджетов tkinter. Ниже перечислены некоторые виджеты и темы, которые будут рассматриваться в данной главе:

- Виджеты `Toplevel` и `Tk`
- Виджеты `Message` и `Entry`
- Виджеты `Checkbutton`, `Radiobutton` и `Scale`
- Изображения: объекты `PhotoImage` и `BitmapImage`
- Параметры настройки виджетов и окон
- Диалоги: стандартные и пользовательские
- Низкоуровневое связывание событий
- Объекты связанных переменных tkinter
- Использование библиотеки Python обработки изображений – расширения PIL (Python Imaging Library) – для работы с изображениями других типов

Глава 9 завершает краткий рассказ, представляя остальные элементы инструментария библиотеки tkinter: меню, текст, холсты, анимацию и другие.

Чтобы сделать этот обзор интереснее, я также попутно введу некоторые идеи повторного использования компонентов. Например, некоторые более поздние примеры будут написаны с использованием компонентов, реализованных для предыдущих примеров. Хотя эти две главы знакомят с интерфейсами, основанными на виджетах, тем не менее данная книга написана также о программировании на языке Python в целом – как будет показано, программирование с использованием библиотеки tkinter в сценариях на языке Python может быть значительно более содержательным, чем просто рисование кружков и стрелок.

Настройка внешнего вида виджетов

До сих пор все кнопки и метки в наших примерах выводились с оформлением по умолчанию, стандартном для соответствующей платформы. В Windows это обычно означает, что они выводятся серым цветом, как

в цветовой схеме, установленной на моем компьютере. Однако библиотека tkinter позволяет придавать виджетам любой другой внешний вид с помощью ряда параметров настройки виджетов и компоновки.

Поскольку обычно я не могу устоять перед соблазном определить для виджетов в примерах собственные настройки, хочу осветить эту тему в самом начале обзора. В примере 8.1 приводятся некоторые параметры настройки, доступные в tkinter.

Пример 8.1. PP4E\Gui\Tour\config-label.py

```
from tkinter import *
root = Tk()
labelfont = ('times', 20, 'bold')      # семейство, размер, стиль
widget = Label(root, text='Hello config world')
widget.config(bg='black', fg='yellow') # желтый текст на черном фоне
widget.config(font=labelfont)          # использовать увеличенный шрифт
widget.config(height=3, width=20)      # начальный размер: строк,символов
widget.pack(expand=YES, fill=BOTH)
root.mainloop()
```

Запомните, что с помощью метода `config` виджета можно в любой момент переопределить значения его параметров, что позволяет не передавать их все конструктору объекта. В данном случае мы используем эту возможность, чтобы установить параметры, которые определяют окно, изображенное на рис. 8.1.

Рис. 8.1. *Нестандартный внешний вид метки*

Если запустить сценарий на компьютере (увы, я не могу показать здесь в цвете, как это выглядит), то вы увидите, что текст метки выводится желтым цветом на черном фоне, причем шрифт сильно отличается от того, что мы до сих пор видели. Этот сценарий настраивает следующие параметры отображения метки:

Цвет

Установкой параметра `bg` метки определяется черный цвет ее фона. Аналогично параметр `fg` изменяет цвет переднего плана (текста) метки на желтый. Эти параметры цвета присутствуют у большинства виджетов tkinter, и в них указывается простое название цвета

(например, 'blue') или шестнадцатеричная строка. Поддерживается большинство знакомых названий цветов (если только вам не довелось работать для компании Crayola[1]). Чтобы более точно определить значение цвета, в этих параметрах можно также передавать строки с шестнадцатеричными значениями – они должны начинаться с символа # и содержать значения насыщенности красного, зеленого и голубого цветов с одинаковым количеством битов для каждого. Например, строка '#ff0000' содержит по восемь битов для каждого цвета и определяет чистый красный цвет – «f» означает в шестнадцатеричном виде четыре единичных бита. Мы еще вернемся к шестнадцатеричному формату, когда встретимся с диалоговым окном выбора цвета далее в этой главе.

Размер

Для метки определяется точный размер в виде количества строк в высоту и символов в ширину путем установки параметров height и width. С помощью этих параметров можно увеличивать размеры метки по сравнению с теми, что устанавливаются менеджером компоновки по умолчанию.

Шрифт

В этом сценарии выбирается нестандартный шрифт для текста метки путем записи в параметр font кортежа из трех элементов, определяющих семейство шрифта, его размер 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"). Библиотека tkinter гарантирует возможность использования названий семейств шрифтов Times, Courier и Helvetica на всех платформах, однако в некоторых системах могут использоваться и другие (например, system – системный шрифт в Windows). Такие настройки шрифта будут действовать для всех виджетов, содержащих текст, например меток, кнопок, полей ввода, списков и Text (последний может одновременно выводить текст, отображаемый различными шрифтами, с помощью «тегов»). Параметр font сохраняет возможность определять шрифт с помощью более старых определений в стиле X Window – длинных строк с дефисами и звездочками, однако более новая форма определения параметров шрифта в виде кортежа более независима от платформы.

Параметры компоновки

Наконец, метка может быть сделана расширяемой и растягиваемой в целом путем установки параметров компоновщика pack, таких как expand и fill, с которыми мы познакомились в предыдущей главе: метка увеличивается вместе с окном. При распахивании окна чер-

[1] Всемирно известный производитель товаров для детского творчества, в том числе цветных карандашей, красок, мелков, фломастеров и пр. – *Прим. ред.*

ный фон заполняет весь экран, а желтый текст помещается в центр – можете попробовать.

В данном сценарии итоговым результатом настроек является метка, по своему внешнему виду коренным образом отличающаяся от тех, что мы создавали до сих пор. Она больше не следует стандартам внешнего вида Windows, но такая согласованность не всегда важна. Для справки отмечу, что библиотека tkinter предоставляет дополнительные способы настройки внешнего вида, не используемые в данном сценарии, но которые могут встретиться вам в других сценариях:

Границы и рельефность

Параметр `bd=N` виджета можно использовать для установки ширины границы, а параметр `relief=S` – ее стиля. *S* может принимать значения `FLAT`, `SUNKEN`, `RAISED`, `GROOVE`, `SOLID` или `RIDGE` – все эти константы экспортирует модуль `tkinter`.

Курсор

Параметр `cursor` позволяет определять внешний вид указателя мыши при наведении его на виджет. Например, `cursor='gumby'` изменяет стрелку на фигурку зеленого человечка. В число других имен указателей, часто используемых в этой книге, входят `watch`, `pencil`, `cross` и `hand2`.

Состояние

Некоторые виджеты поддерживают понятие состояния, влияющее на их внешний вид. Например, виджет с параметром `state=DISABLED` обычно рисуется на экране закрашенным (окрашивается в серый цвет) и делается неактивным. Значение `NORMAL` делает его обычным. Некоторые виджеты поддерживают также состояние `READONLY`, когда сам виджет отображается, как обычно, но он никак не откликается на попытки изменения.

Отступы (padding)

Вокруг многих виджетов (кнопок, меток и текста) можно добавить дополнительное пустое пространство с помощью параметров `padx=N` и `pady=N`. Интересно, что эти параметры можно определять и в вызовах метода `pack` (тогда пустое пространство добавляется вокруг виджета в целом), и в самом объекте виджета (в результате увеличивается сам графический элемент).

Чтобы проиллюстрировать некоторые из этих дополнительных настроек, в примере 8.2 создается кнопка, изображенная на рис. 8.2, и изменяется форма указателя мыши, когда он помещается над кнопкой.

Пример 8.2. PP4E\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()
```

***Рис. 8.2.** Параметры кнопки в действии*

Чтобы увидеть эффект, создаваемый этими двумя параметрами, попробуйте поиграть с ними на своем компьютере. Большинству виджетов можно придать новый внешний вид таким же способом, и в этой книге мы будем неоднократно встречаться с такими параметрами. Мы встретимся также с операционными параметрами, такими как `focus` (передает фокуса ввода), и другими. У виджетов могут быть десятки параметров, большинство из которых имеет разумные значения по умолчанию, создающие принятый на каждой оконной платформе внешний вид, что является одной из причин простоты использования tkinter. Но при необходимости tkinter позволяет создавать значительно более индивидуальные изображения.

Дополнительные способы применения параметров настройки, обеспечивающих типичный внешний вид виджетов, можно увидеть в разделе «Настройка виджетов с помощью классов» в предыдущей главе и особенно в примерах `ThemedButton`. Теперь, когда вы больше знаете о настройках, вам будет проще понять, как настройки в этих примерах, выполняемые в подклассах виджетов, наследуются всеми экземплярами и подклассами. Новое расширение `ttk`, описываемое в главе 7, также реализует дополнительные способы настройки виджетов, вводя понятие тем оформления. Больше подробностей и ссылки на ресурсы, посвященные `ttk`, вы найдете в предыдущей главе.

Окна верхнего уровня

Графические интерфейсы, построенные на базе tkinter, всегда имеют 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 размещения наиболее важных виджетов. Помимо этого окна сценарии, использующие библиотеку tkinter, могут порождать любое число независимых окон, которые создаются и открываются по требованию, в результате создания объектов виджетов `Toplevel`.

Каждый объект `Toplevel` порождает на экране новое окно и автоматически добавляет его в поток обработки цикла событий программы (для активации новых окон не нужно вызывать метод `mainloop`). В примере 8.3 создается корневое окно и два дополнительных окна.

Пример 8.3. PP4E\Gui\Tour\toplevel0.py

```
import sys
from tkinter import Toplevel, Button, Label

win1 = Toplevel() # два независимых окна
win2 = Toplevel() # являющихся частью одного и того же процесса

Button(win1, text='Spam', command=sys.exit).pack()
Button(win2, text='SPAM', command=sys.exit).pack()

Label(text='Popups').pack() # по умолчанию добавляется в корневое окно Tk()
win1.mainloop()
```

Сценарий `toplevel0` получает корневое окно по умолчанию (к которому прикрепляется метка `Label`, потому что для нее не указан родитель) и создает два самостоятельных окна `Toplevel`, которые появляются и действуют независимо от корневого окна, как показано на рис. 8.3.

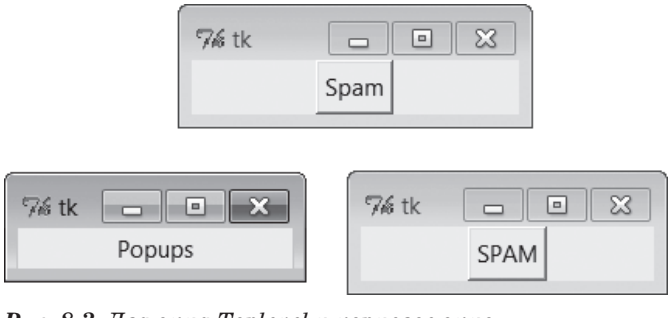


***Рис. 8.3.** Два окна Toplevel и корневое окно*

Два окна `Toplevel` в правой части являются полноценными окнами – они могут независимо сворачиваться, распахиваться на весь экран и так далее. Обычно окна `Toplevel` используются для реализации многооконных интерфейсов, а также модальных и немодальных диалогов (подробнее о диалогах рассказывается в следующем разделе). Они сохраняются до тех пор, пока не будут явно закрыты или пока создавшее их приложение не завершит работу.

Согласно реализации примера щелчок на кнопке с крестиком в правом верхнем углу любого из окон Toplevel закрывает только это окно. С другой стороны, щелчок на любой из кнопок или на крестике главного окна закроет все остальные и завершит программу (подробнее о протоколе завершения рассказывается чуть ниже).

Важно знать, что, хотя окна Toplevels и действуют независимо, они не являются независимыми процессами – если программа завершится, автоматически будут закрыты все ее окна, включая все окна Toplevel, которые она могла создать. Позднее будет показано, как обойти это правило путем запуска независимых программ с графическим интерфейсом.

Виджеты Toplevel и Tk

Окно Toplevel напоминает Frame тем, что отщепляется в самостоятельное окно и обладает дополнительными методами, позволяющими работать со свойствами окна верхнего уровня. Виджет Tk в общем похож на виджет Toplevel, но используется для представления корневого окна приложения. Окна Toplevel имеют родителя, тогда как окно Tk – нет. Оно является настоящим корнем иерархии виджетов, создаваемых при конструировании графических интерфейсов с помощью библиотеки tkinter.

В примере 8.3 корневое окно Tk было получено даром, потому что для виджета Label был использован родитель по умолчанию, назначаемый при отсутствии первого аргумента в вызове конструктора:

```
Label(text='Popups').pack() # корневое окно Tk() по умолчанию
```

Передача значения None в первом аргументе конструктора виджета (или в именованном аргументе master) также приводит к назначению родителя по умолчанию. В других сценариях корневое окно Tk создается явно, например:

```
root = Tk()
Label(root, text='Popups').pack() # явное создание корневого окна Tk()
root.mainloop()
```

В действительности, из-за того, что графические интерфейсы tkinter строятся в виде иерархии, по умолчанию *всегда* создается хотя бы одно корневое окно Tk, явно, как в данном примере, или нет. Хотя это и не типично, тем не менее в приложении вручную может создаваться несколько корневых окон Tk, при этом программа завершается только после закрытия всех окон Tk. Первое созданное корневое окно Tk – явно, в программном коде, или автоматически, интерпретатором, – используется как родитель по умолчанию для виджетов и других окон, при создании которых родитель не указывается.

В целом корневое окно Tk должно использоваться для отображения какой-либо информации верхнего уровня. Если не прикрепить графические элементы к корневому окну, при запуске сценария оно будет вы-

ведено как странное пустое окно (часто это происходит из-за забывчивости, когда программист создает виджеты, использующие родителя по умолчанию, но забывает вызвать метод компоновщика, выполняющий размещение виджетов). Технически можно подавить создание корневого окна по умолчанию и создать несколько корневых окон с помощью виджета Tk, как показано в примере 8.4.

Пример 8.4. PP4E\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()
```

Если запустить этот сценарий, он создаст только два окна, изображенные на рис. 8.3 (третье корневое окно не будет создано). Но чаще корневой объект Tk используется как главное окно, а виджеты Toplevel – как всплывающие окна приложения.

Обратите внимание: чтобы закрыть только одно окно, вместо функции sys.exit, которая завершает работу всей программы, вызывается метод destroy этого окна – чтобы понять, как действует этот метод, перейдем к изучению протоколов окна.

Протоколы окна верхнего уровня

Виджеты Tk и Toplevel экспортируют дополнительные методы и свойства, предназначенные для той роли, которую они играют на верхнем уровне, что иллюстрируется примером 8.5.

Пример 8.5. PP4E\Gui\Tour\toplevel2.py

```
"""
Открывает три новых окна со стилями
метод destroy() закрывает одно окно, метод quit() закрывает все окна и завершает
приложение (прерывает работу функции mainloop);
окна верхнего уровня имеют заголовки, значки, могут сворачиваться
и восстанавливаться и поддерживают протокол событий wm;
приложение всегда имеет корневое окно, создаваемое по умолчанию или явно,
вызовом конструктора Tk(); все окна верхнего уровня являются контейнерами,
но они никогда не размещаются с помощью менеджера компоновки; объект Toplevel
напоминает фрейм Frame, но в действительности является новым окном и может иметь
собственное меню;
"""

from tkinter import *
```

```
root = Tk() # explicit 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...')                                # установка границ
    win.protocol('WM_DELETE_WINDOW', lambda:None)       # игнорировать закрытие
    win.iconbitmap('py-blue-trans-out.ico')             # вместо значка Tk

    msg = Button(win, text=tree, command=win.destroy)  # закрывает 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() # завершает программу
root.mainloop()
```

Эта программа добавляет виджеты в корневое окно `Tk`, сразу выводит три окна `Toplevel` с прикрепленными к ним кнопками и использует специальные протоколы верхнего уровня. Если запустить этот пример, он создаст картинку, переданную в черно-белом изображении на рис. 8.4 (на мониторе текст кнопок отображается синим, зеленым и красным цветом).

***Рис. 8.4.** Три настроенных окна Toplevel*

Здесь следует отметить несколько деталей, касающихся функционирования, которые станут более заметными, если вы запустите сценарий на своем компьютере:

Перехват закрытия: `protocol`

Этот сценарий перехватывает событие закрытия окна менеджера окон с помощью метода виджета верхнего уровня `protocol`, поэтому при нажатии кнопки `X` в правом верхнем углу какого-либо из трех окон `Toplevel` ничего не происходит. Строка `WM_DELETE_WINDOW` обозначает операцию закрытия. С помощью этого метода можно запретить закрытие окон, кроме как из создаваемых в сценарии виджетов. Создаваемая этим сценарием функция `lambda: None` лишь возвращает значение `None` и больше ничего не делает.

Закрытие одного окна (и его дочерних окон): `destroy`

При нажатии на большую черную кнопку в любом из трех дополнительных окон закрывается только это окно, потому что это действие вызывает метод `destroy` виджета. Остальные окна продолжают существовать, как это свойственно диалоговым окнам. Технически вызов этого метода приводит к уничтожению соответствующего виджета и всех остальных виджетов, для которых он является родителем. В случае окон под этим подразумевается все их содержимое. В случае более простых виджетов метод `destroy` уничтожает сам виджет.

Вследствие того, что окна `Toplevel` имеют родителя, эти их отношения могут иметь последствия при применении метода `destroy`. Уничтожение окна, даже первого корневого окна `Tk`, созданного автоматически или явно, – которое является родителем по умолчанию, – приводит к уничтожению всех его дочерних окон. Так как корневые окна `Tk` не имеют родителя, на них никак не действует уничтожение других окон. При этом уничтожение последнего (или единственного) корневого окна `Tk` приводит к завершению программы. Окна `Toplevel` всегда уничтожаются при уничтожении родителя, но их уничтожение никак не влияет на другие окна, для которых они не являются предками. Это делает их идеальными для создания диалогов. Технически виджет `Toplevel` может быть дочерним по отношению к любому виджету и автоматически будет уничтожен вместе с родителем, однако обычно они создаются как потомки окна `Tk`, созданного явно или автоматически.

Закрытие всех окон: `quit`

Чтобы закрыть сразу все окна и завершить приложение с графическим интерфейсом (в действительности – активный вызов `mainloop`), кнопка корневого окна вызывает метод `quit`. То есть нажатие кнопки в корневом окне завершает работу приложения. Метод `quit` немедленно завершает приложение в целом и закрывает все его окна. Он может быть вызван относительно любого виджета tkinter, а не только относительно окна верхнего уровня – этот метод имеется также у фреймов, кнопок и других виджетов. Дополнительные подробности о методах `quit` и `destroy` вы найдете в обсуждении метода `bind` и его события `<Destroy>` далее в этой главе.

Заголовки окон: `title`

В главе 7 говорилось о методе `title` виджетов окон верхнего уровня (`Tk` и `Toplevel`), позволяющем изменять текст, выводимый в области верхней кромки окна. В данном случае в качестве текста заголовка окна устанавливается строка `'Si...'`, замещающая текст по умолчанию `'tk'`.

Значки окон: `iconbitmap`

Метод `iconbitmap` изменяет значок окна верхнего уровня. Он принимает значок или файл с растровым изображением и использует его в качестве графического значка окна, когда оно сворачивается и открывается. Если в Windows передать имя файла с расширением *.ico* (в данном примере используется такой файл, находящийся в текущем каталоге), он заменит значок по умолчанию с красными буквами «Tk», который обычно появляется в левом верхнем углу окна, а также в панели задач Windows. На других платформах вам может потребоваться использовать иные соглашения о файлах со значками, если вызов этого метода в наших примерах не дает желаемого результата (или просто закомментируйте вызов этого метода, если он приводит к аварийному завершению сценариев) – значки обычно являются платформозависимой особенностью, работа с ними зависит от используемого менеджера окон.

Управление компоновкой

Окна верхнего уровня служат контейнерами для других виджетов, подобно отдельному фрейму `Frame`. Однако в отличие от фреймов, виджеты – окна верхнего уровня – сами не компонуются (и не размещаются каким-либо другим менеджером компоновки). Для встраивания виджетов этот сценарий передает свои окна в аргументах, определяющих родительское окно, конструкторам меток и кнопок.

Имеется также возможность определять максимальный размер окна (физические размеры экрана в виде кортежа [ширина, высота]) с помощью метода `maxsize()` и устанавливать начальные размеры окна с помощью высокоуровневого метода `geometry("` *`width x height + x + y`* `")`. На практике гораздо проще и удобнее позволить библиотеке tkinter (или вашим пользователям) самой устанавливать размер окон, тем не менее размер экрана может пригодиться при выборе масштаба отображения изображений (смотрите обсуждение PyPhoto в главе 11, например).

Кроме того, графические элементы окон верхнего уровня поддерживают другие типы протоколов, которые будут позднее использованы в данном обзоре:

Состояние

Методы `iconify` и `withdraw` объектов окон верхнего уровня позволяют сценариям сворачивать или удалять окна на лету; метод `deiconify` перерисовывает свернутое или удаленное окно. Метод `state` возвра-

щает или изменяет состояние окна – допустимыми значениями, которые могут устанавливаться или возвращаться, являются: `iconic`, `withdrawn`, `zoomed` (в Windows: распахнутое на весь экран с помощью `geometry` или какого-либо другого метода) и `normal` (достаточно большого размера, чтобы вместить все содержимое). Методы `lift` и `lower` поднимают или опускают окно относительно других (метод `lift` является аналогом команды `raise` библиотеки Tk, которое является зарезервированным словом в языке Python). Их использование демонстрируется в сценариях будильника в конце главы 9.

Меню

Каждое окно верхнего уровня может иметь собственное меню – виджеты `Tk` и `Toplevel` принимают параметр `menu`, который используется для подключения горизонтальной строки меню с открывающимися списками элементов выбора. Эта строка меню выглядит соответствующим образом на каждой платформе, где выполняется сценарий. Меню будут изучаться в начале главы 9.

Большую часть методов окна верхнего уровня, используемых для взаимодействия с менеджером окон, можно также вызывать под именами с префиксом «wm_». Например, методы `state` и `protocol` можно также вызвать как `wm_state` и `wm_protocol`.

Обратите внимание, что в примере 8.3 при вызове конструктора `Toplevel` ему явно передается родительский виджет – корневое окно `Tk` (то есть `Toplevel(root)`). Окна `Toplevel` можно связывать с родительскими, как и любые другие виджеты, хотя зрительно они не встраиваются в родительские окна. Такой способ написания сценария имел целью избежать одной, на первый взгляд странной, особенности. Если бы окно создавалось так:

```
win = Toplevel()                        # новое окно
```

и корневое окно `Tk` при этом еще не существовало бы, этот вызов создал бы корневое окно `Tk` по умолчанию, которое стало бы родителем для окон `Toplevel`, как при всяком другом вызове графического элемента без передачи аргумента со ссылкой на родителя. Проблема в том, это делает принципиальным местоположение следующей строки:

```
root = Tk()                             # явное создание корня
```

Если поместить эту строку выше вызовов конструктора `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 8.4. 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 виджетах окон верхнего уровня, но часто доступ к ним можно получить через атрибут `master` виджета, хранящего ссылку на родительское окно. Например, изменение заголовка окна, в котором содержится фрейм, можно реализовать так:

```
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, реализуемые в виде классов, должны оставить право на установку свойств окон за своим приложениями-клиентами.

Для виджетов верхнего уровня существуют другие инструменты, некоторые из которых могут не встретиться в этой книге. Например, в менеджерах окон Unix можно также устанавливать имя значка окна (`iconname`). Поскольку некоторые параметры значков можно применять только в сценариях, выполняемых в Unix, подробности, касающиеся этой темы, смотрите в других ресурсах по библиотекам 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 с целью показать или запросить дополнительную информацию. Существует два вида диалогов: модальные и немодальные:

Модальные

Эти диалоги блокируют остальную часть интерфейса, пока окно диалога не будет закрыто – выполнение программы будет продолжено после получения диалогом ответа пользователя.

Немодальные

Эти диалоги могут оставаться на экране неопределенное время, не создавая 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. Существует три основных способа вывести диалог с помощью библиотеки tkinter: вызовом стандартных диалогов, обращением к современному объекту `Dialog` и путем создания пользовательских диалоговых окон с помощью `Toplevel` и других типов виджетов. Рассмотрим основы использования всех трех схем.

Стандартные (типичные) диалоги

Вызовы стандартных диалогов проще, поэтому начнем с них. В составе библиотеки tkinter поставляется набор готовых диалогов, реализующих многие из наиболее часто встречающихся окон, генерируемых программами, – диалоги выбора файла, диалоги с сообщениями об ошибках и предупреждениями и диалоги, позволяющие запросить ввод данных. Они называются *стандартными диалогами*, поскольку входят в состав библиотеки tkinter и используют библиотечные вызовы для конкретных платформ, чтобы принять вид, свойственный данной платформе. Например, диалог открытия файла в библиотеке tkinter выглядит как любой другой подобный диалог в Windows.

Все стандартные диалоги являются модальными (они не возвращают управление, пока пользователь не закроет диалог) и блокируют главное окно программы. Сценарии могут настраивать окна этих диалогов, передавая текст сообщения, заголовки и тому подобное. Они очень просты в использовании, поэтому сразу перейдем к примеру 8.6 (который хранится в файле с расширением *.pyw*, чтобы подавить вывод окна консоли в Windows при запуске сценария щелчком мыши):

Пример 8.6. PP4E\Gui\Tour\dlg1.pyw

```
from tkinter import *
from tkinter.messagebox 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()
```

Анонимная `lambda`-функция использована здесь в качестве оболочки вызова `showerror`, для передачи двух жестко определенных аргументов (напомню, что обработчики событий не получают аргументов от самой библиотеки tkinter). Если запустить этот сценарий, он создаст главное окно, изображенное на рис. 8.5.

Нажатие кнопки Quit в этом окне выводит диалог (рис. 8.6) – вызовом стандартной функции `askyesno` из модуля `messagebox`, входящего в состав пакета `tkinter`. В Unix и Macintosh этот диалог выглядит иначе, а в Windows выглядит, как показано на рисунке (на практике внешний вид диалога зависит от версии и настроек Windows – в моей системе Window 7 с настройками по умолчанию он выглядит несколько иначе, чем в Windows XP, как было показано в предыдущем издании).

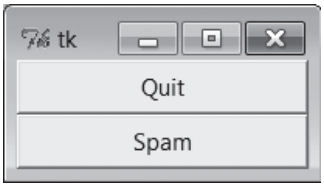


Рис. 8.5. *Главное окно dlg1: кнопки вызывают появление дополнительных окон*

Диалог на рис. 8.6 блокирует программу, пока пользователь не щелкнет по одной из кнопок – при выборе кнопки Yes (или нажатии клавиши Enter) вызов диалога возвращает значение `True`, и сценарий выводит стандартный диалог `showwarning` (рис. 8.7), вызывая функцию `showwarning`.

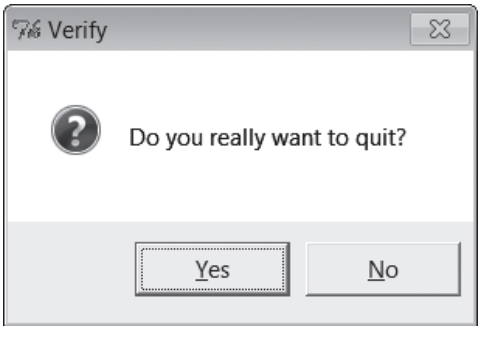


Рис. 8.6. *Диалог askyesno, выводимый сценарием dlg1 (в Windows 7)*

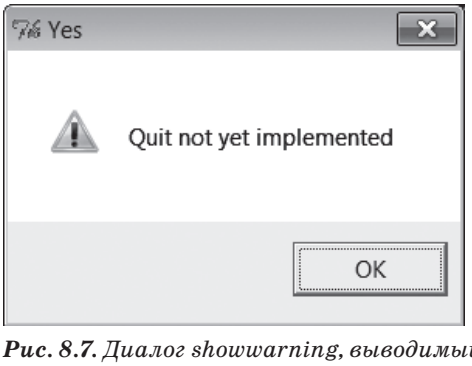


Рис. 8.7. *Диалог showwarning, выводимый сценарием dlg1*

В диалоге на рис. 8.7 пользователь может только нажать кнопку OK. Если щелкнуть на кнопке No в диалоге на рис. 8.6, вызов `showinfo` создаст соответствующее окно диалога (рис. 8.8). Наконец, если в главном окне щелкнуть по кнопке Spam, то с помощью стандартного вызова `showerror` будет создан стандартный диалог `showerror` (рис. 8.9).

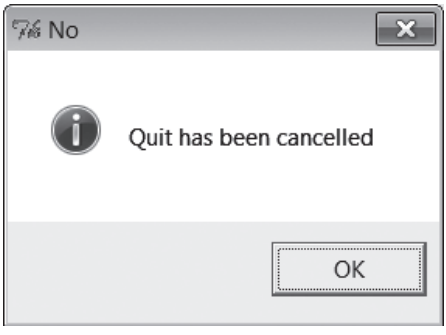


Рис. 8.8. *Диалог showinfo, выводимый сценарием dlg1*

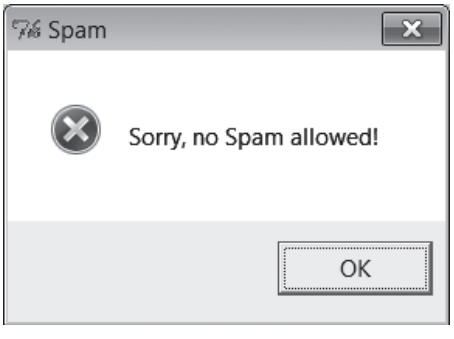


Рис. 8.9. *Диалог showerror, выводимый сценарием dlg1*

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 разработки и обеспечивают привычный внешний вид.

«Умная» и многократно используемая кнопка Quit

Для некоторых из этих готовых диалогов можно найти лучшее применение. В примере 8.7 реализована прикрепляемая кнопка `Quit`, которая с помощью стандартных диалогов получает подтверждение в ответ на запрос о завершении. Поскольку она реализована в виде класса, ее можно прикреплять и повторно использовать в любом приложении, где требуется кнопка `Quit` с запросом на подтверждение. Так как в этой кнопке использованы стандартные диалоги, она должным образом выглядит на любой платформе.

Пример 8.7. PP4E\Gui\Tour\quitter.py

```
"""
кнопка Quit, которая запрашивает подтверждение на завершение;
для повторного использования достаточно прикрепить экземпляр к другому
графическому интерфейсу и скомпоновать с желаемыми параметрами
"""

from tkinter import *                       # импортировать классы виджетов
from tkinter.messagebox import askokcancel # импортировать стандартный диалог

class Quitter(Frame):                       # подкласс графич. интерфейса
    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, expand=YES, fill=BOTH)

    def quit(self):
        ans = askokcancel('Verify exit', "Really quit?")
        if ans: Frame.quit(self)

if __name__ == '__main__': Quitter().mainloop()
```

Вообще этот модуль предназначен для использования в других программах, но может запускаться самостоятельно и тогда выводит кнопку, которая в нем реализована. На рис. 8.10 слева вверху показана сама кнопка Quit и диалог `askokcancel` запроса подтверждения, выведенный при нажатии кнопки Quit.

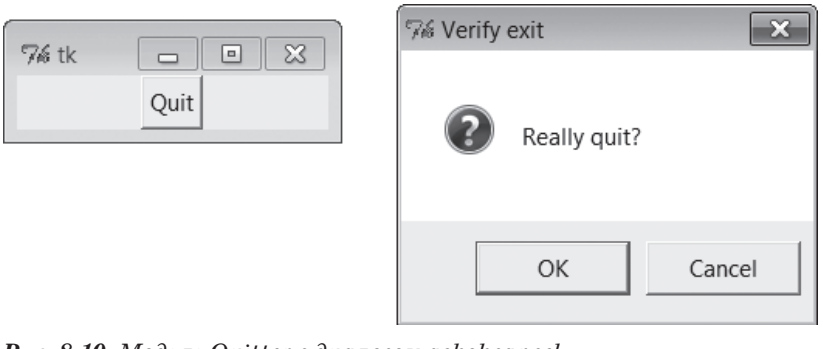


***Рис. 8.10.** Модуль Quitter с диалогом askokcancel*

Если нажать кнопку OK в этом окне, модуль `Quitter` вызовет метод `quit` элемента `Frame` и закроет графический интерфейс, к которому прикреплена кнопка (на самом деле завершит работу функции `mainloop`). Но чтобы действительно оценить пользу, приносимую такими подпружиненными кнопками, рассмотрим графический интерфейс 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 снимках с экрана, а напишем на языке Python демонстрационный сценарий, который будет генерировать их по требованию. Ниже приводится один из способов сделать это. Во-первых, напишем модуль, приведенный в примере 8.8, который определяет таблицу соответствий между именами демонстрационных программ и вызовами стандартных диалогов (будем использовать `lambda`-выражения для обертывания вызовов вызова, если функции диалога нужно передать дополнительные аргументы).

Пример 8.8. PP4E\Gui\Tour\dialogTable.py

```
# определяет таблицу имя:обработчик с демонстрационными примерами

from tkinter.filedialog   import askopenfilename # импортировать стандартные
from tkinter.colorchooser import askcolor        # диалоги из Lib\tkinter
from tkinter.messagebox   import askquestion, showerror
from tkinter.simpledialog 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 текст в `stdout`). Затем напишем сценарий на языке Python, представленный в примере 8.9, который просто генерирует кнопки для всех этих элементов таблицы – использует ее ключи как метки кнопок, а значения как обработчики событий для кнопок.

Пример 8.9. PP4E\Gui\Tour\demoDlg.py

```
"создает панель с кнопками, которые вызывают диалоги"

from tkinter import *         # импортировать базовый набор виджетов
from dialogTable import demos # обработчики событий для кнопок
from quitter import Quitter   # прикрепить к себе объект quit

class Demo(Frame):
    def __init__(self, parent=None, **options):
        Frame.__init__(self, parent, **options)
        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()
```

Если запустить этот пример как самостоятельный сценарий, он создаст окно, изображенное на рис. 8.11: это панель демонстрационных кнопок, при нажатии которых просто выполняется передача управления в соответствии со значениями в таблице из модуля `dialogTable`.

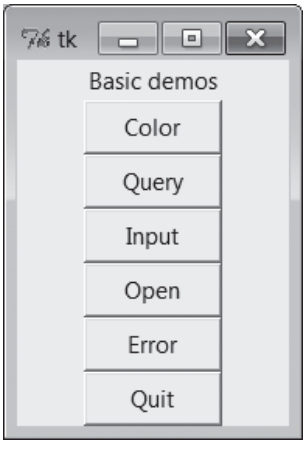


Рис. 8.11. *Главное окно demoDlg*

Обратите внимание, что этот сценарий управляется содержимым словаря из модуля `dialogTable`, поэтому мы можем изменять набор кнопок, изменяя только `dialogTable` (никакого выполняемого программного кода в `demoDlg` менять не нужно). Отметьте также, что кнопка Quit является в данном случае прикрепленным экземпляром класса `Quitter` из предыдущего раздела, причем она скомпонована с теми же параметрами, что и остальные кнопки, – по крайней мере эту часть программного кода уже не нужно будет писать снова.

Кроме всего прочего этот класс обеспечивает передачу любых именованных аргументов `**options` конструктору своего суперкласса `Frame`. Хотя в данном примере эта возможность и не используется, тем не менее вызывающие программы могут передавать параметры настройки во время создания экземпляра (`Demo(o=v)`), вместо того чтобы выполнять настройку позднее (`d.config(o=v)`). В этом нет особой необходимости, но такая реализация обеспечивает возможность использования класса `Demo` как обычного виджета фрейма (в чем, собственно, и заключается прием создания подклассов). Позднее мы увидим, как можно использовать эту особенность.

Мы уже видели некоторые диалоги, запускаемые другими кнопками этой демонстрационной панели, поэтому я коснусь здесь только новых.

Например, нажатие кнопки Query генерирует стандартный диалог, изображенный на рис. 8.12.

Этот диалог askquestion выглядит как askyesno, который мы видели раньше, но в действительности возвращает строку "yes" или "no" (askyesno и askokcancel возвращают True или False). Нажатие кнопки Input генерирует стандартный диалог askfloat, изображенный на рис. 8.13.

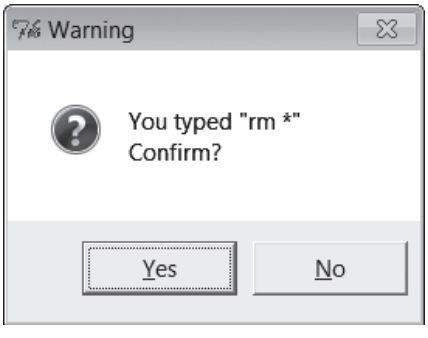


Рис. 8.12. *Запрос demoDlg, диалог askquestion*

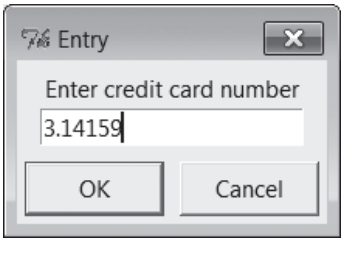


Рис. 8.13. *Ввод demoDlg, диалог askfloat*

Прежде чем вернуть управление, этот диалог автоматически проверяет введенные данные на соответствие синтаксису записи чисел с плавающей точкой; он является представителем группы диалогов 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 на кнопке Cancel. Два родственных ему диалога возвращают объекты целого числа и строки.

При нажатии кнопки Open мы получаем стандартный диалог открытия файла, создаваемый вызовом функции askopenfilename и изображенный на рис. 8.14. Это внешний вид в Windows 7 – в Mac OS, Linux и в более старых версиях Windows этот диалог может выглядеть совершенно иначе.

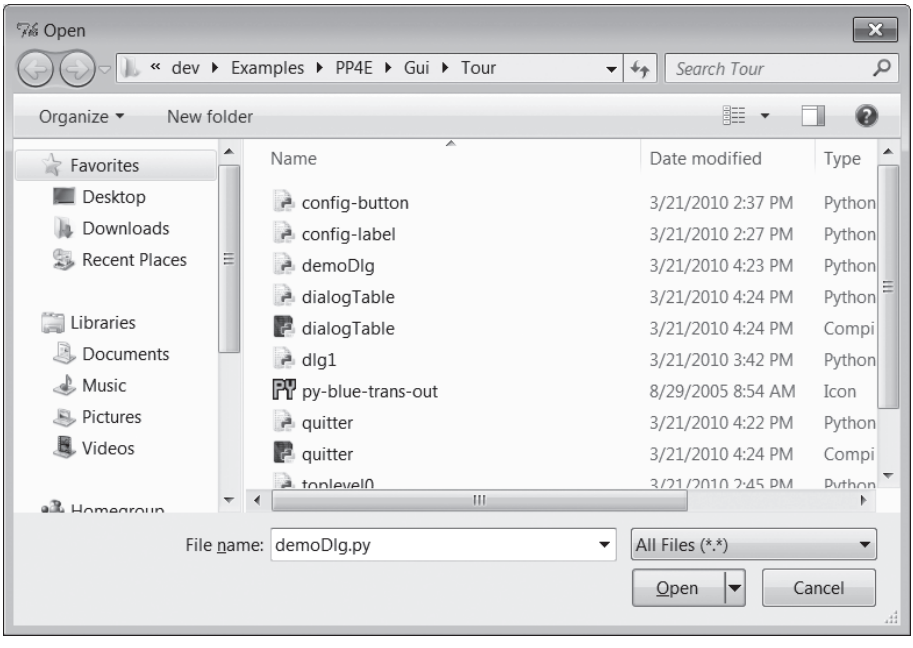


***Рис. 8.14.** Открытие файла в demoDlg, диалог askopenfilename*

Аналогичный диалог выбора имени сохраняемого файла создается вызовом функции `asksaveasfilename` (пример можно найти в разделе, посвященном виджету `Text`, в главе 9). Оба файловых диалога дают пользователю возможность перемещаться по файловой системе для выбора нужного имени файла, которое возвращается вместе с полным путем к файлу при нажатии кнопки Open; если была нажата кнопка Cancel, возвращается пустая строка. Оба диалога поддерживают дополнительные протоколы, не показанные в этом примере:

- Им можно передать именованный аргумент `filetypes` – группу шаблонов имен для выбора файлов, появляющихся в раскрывающемся списке в нижней части диалога.
- Им можно передать параметры `initialdir` (начальный каталог), `initialfile` (для поля ввода File name), `title` (заголовок окна диалога), `defaultextension` (расширение, добавляемое, когда у выбранного файла нет расширения) 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 – эту особенность мы будем использовать в последующих примерах.

В модуле `filedialog`, в библиотеке tkinter, имеется еще один часто используемый диалог, вызываемый функцией `askdirectory`, который мо-

жет использоваться, чтобы дать пользователю возможность выбрать каталог. Он выводит структуру каталогов в виде дерева – пользователь может перемещаться по этому дереву и выбирать нужный ему каталог. Эта функция принимает именованные аргументы, включая `initialdir` и `title`. Для сохранения имени последнего выбранного каталога, который будет автоматически открыт при следующем вызове диалога, можно использовать соответствующий объект `Directory`.

Большинство из этих интерфейсов позднее будут использованы в книге, особенно для реализации диалогов выбора файлов в приложении PyEdit, в главе 11, но вы можете, забежав вперед, узнать дополнительные подробности прямо сейчас. Диалог выбора каталога будет показан в примере приложения PyPhoto, в главе 11, и в примере приложения PyMailGUI, в главе 14 – опять же, вы можете забежать вперед, чтобы посмотреть примеры программного кода и снимки с экрана.

Наконец, кнопка `Color` вызывает стандартную функцию `askcolor`, которая генерирует стандартный диалог выбора цвета, изображенный на рис. 8.15.

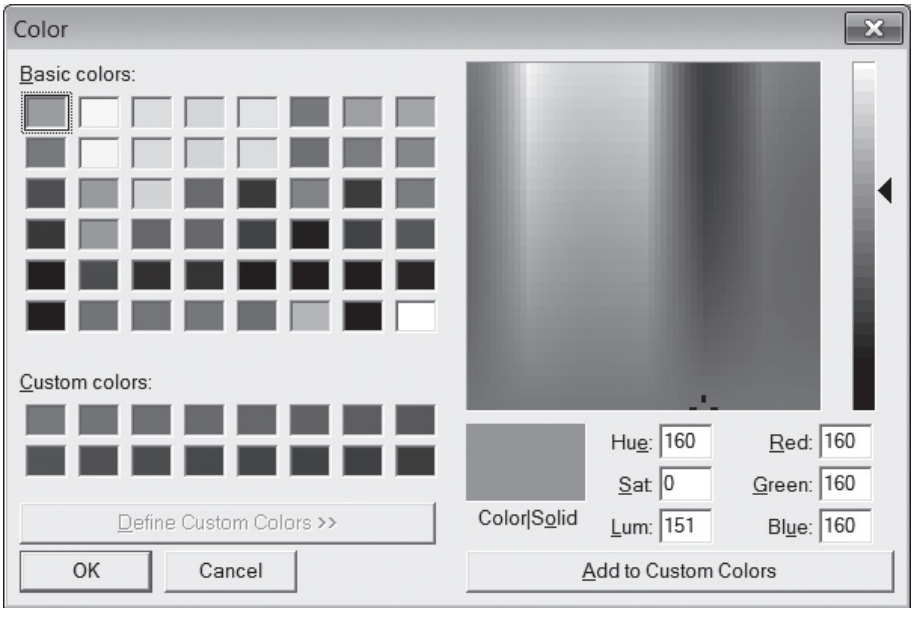


Рис. 8.15. *Выбор цвета в demoDlg, диалог askcolor*

При нажатии в нем кнопки `OK` возвращается структура данных, идентифицирующая выбранный цвет, которую можно использовать в любом контексте tkinter, где требуется указать цвет. В нее входят значения RGB и шестнадцатеричная строка цвета (например, `((160, 160, 160), '#a0a0a0')`). Подробнее об использовании этого набора будет рассказы-

ваться несколько позже. При нажатии кнопки Cancel диалог возвращает кортеж, состоящий из двух значений None.

Вывод результатов, возвращаемых диалогами, и передача данных обработчикам с помощью lambda-выражений

Демонстрационная панель запуска диалогов выводит стандартные диалоги и может быть использована для вывода других диалогов простым изменением импортируемого модуля dialogTable. Однако в существующем виде этот пример только показывает диалоги, и было бы неплохо посмотреть на возвращаемые ими значения, чтобы знать, как использовать их в сценариях. В примере 8.10 добавлен вывод результатов стандартных диалогов в стандартный поток вывода stdout.

Пример 8.10. PP4E\Gui\Tour\demoDlg-print.py

```
"""
то же, что и предыдущий пример, но выводит значения, возвращаемые диалогами;
lambda-выражение сохраняет данные из локальной области видимости для передачи их
обработчику (обработчик события нажатия кнопки обычно не получает аргументов,
а автоматические ссылки в объемлющую область видимости некорректно работают
с переменными цикла) и действует подобно вложенной инструкции def, такой как:
def func(key=key): self.printit(key)
"""

from tkinter import *                   # импортировать базовый набор виджетов
from dialogTable import demos           # обработчики событий от кнопок
from quitter import Quitter             # прикрепить к себе объект quit

class Demo(Frame):
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        Label(self, text="Basic demos").pack()
        for key in demos:
            func = (lambda key=key: self.printit(key))
            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]()) # извлечь, вызвать, вывести

if __name__ == '__main__': Demo().mainloop()
```

Этот сценарий создает то же самое главное окно панели кнопок, но обратите внимание, что обработчик события теперь является анонимной функцией, созданной с помощью lambda-выражения, а не прямой ссылкой на вызов диалога в словаре demos, импортированном из модуля dialogTable:

```
# задействовать поиск значения в объемлющей области видимости
func = (lambda key=key: self.printit(key))
```

Мы уже говорили о такой возможности в предыдущей главе, но здесь мы впервые использовали lambda-выражение подобным образом, поэтому разберемся в том, что происходит. Так как обработчики событий нажатий кнопок вызываются без аргументов, то при необходимости передать обработчику *дополнительные данные* для него нужно создать оболочку в виде объекта, который запомнит эти дополнительные данные и передаст их фактическому обработчику. В данном случае при нажатии кнопки вызывается функция, создаваемая lambda-выражением, – промежуточная функция, сохраняющая информацию из объемлющей области видимости. Благодаря этому действительный обработчик, printit, получит дополнительный аргумент name и выполнит действия, связанные с нажатой кнопкой, несмотря на то, что этот аргумент не был передан самой библиотекой tkinter. Фактически lambda-выражение сохраняет и передает информацию о состоянии.

Заметьте, однако, что в теле функции, создаваемой этим lambda-выражением, используются ссылки на значения self и key, находящиеся в объемлющей области видимости. Во всех последних версиях Python ссылка на self действует автоматически, в соответствии с правилами поиска значений в объемлющих областях видимости, но значение key необходимо передать явно, в виде *аргумента со значением по умолчанию*, иначе все функции, сгенерированные lambda-выражением, получат одно и то же значение – которое получит переменная key в последней итерации цикла. Как мы узнали в главе 7, ссылки на переменные в объемлющей области видимости разрешаются в момент вызова вложенной функции, а ссылки на значения по умолчанию – в момент создания вложенной функции. Так как значение self не изменится после создания функции, мы можем довериться правилам поиска только этого имени, но не переменной цикла key.

В прежних версиях Python требовалось явно передавать любые значения из объемлющей области видимости в виде аргументов со значениями по умолчанию, используя любой из двух следующих приемов:

```
# использовать простые аргументы со значениями по умолчанию
func = (lambda self=self, name=key: self.printit(name))

# использовать связанный метод по умолчанию
func = (lambda handler=self.printit, name=key: handler(name))
```

В настоящее время для получения значения self можно использовать более простой прием автоматических ссылок в объемлющую область видимости, однако для передачи значения переменной key по-прежнему требуется использовать значение по умолчанию аргумента (а кроме того, передачу данных в виде значений по умолчанию можно встретить в давно написанных сценариях на языке Python).

Обратите внимание, что круглые скобки вокруг lambda-выражений здесь не являются обязательными – я добавляю их, потому что предпочитаю визуально отделять lambda-выражения от окружающего программного

кода (ваши предпочтения могут отличаться от моих). Отметьте также, что здесь lambda-выражение можно заменить вложенной инструкцией def. Однако в отличие от инструкции def lambda-выражение может появляться внутри вызова конструктора Button, потому что это выражение и ему не требуется присваивать имя. Следующие две формы совершенно равноценны:

```
for (key, value) in demos.items():
    func = (lambda key=key: self.printit(key)) # может вкладываться в вызов
                                               # Button()
for (key, value) in demos.items():
    def func(key=key): self.printit(key)       # а инструкция def - нет
```

Здесь можно также использовать вызываемый объект класса, который сохраняет состояние в виде атрибутов экземпляра (смотрите подсказку в учебном примере __call__ главы 7). Но как правило, если нужно, чтобы результат lambda-выражения в последующих вызовах использовал переменные из объемлющей области, просто используйте их имена и позвольте интерпретатору самому сохранять значения для последующего использования или передайте их в качестве значений по умолчанию, чтобы обеспечить сохранение значений на этапе создания функции. Последний способ необходим, только если используемая переменная может изменить значение перед тем, как произойдет вызов обработчика.

Если запустить этот сценарий, он создаст то же окно (рис. 8.11) и дополнительно будет выводить значения, возвращаемые диалогами, в стандартный поток вывода. Ниже приводится вывод сценария после щелчков мышью на всех кнопках в главном окне и выбора в каждом диалоге обеих кнопок Cancel/No и OK/Yes:

```
C:\...\PP4E\Gui\Tour> python demoDlg-print.py
Color returns => (None, None)
Color returns => ((128.5, 128.5, 255.99609375), '#8080ff')
Query returns => no
Query returns => yes
Input returns => None
Input returns => 3.14159
Open returns =>
Open returns => C:/Users/mark/Stuff/Books/4E/PP4E/dev/Examples/PP4E/Launcher.py
Error returns => ok
```

Теперь, когда были показаны результаты вызова всех диалогов, я хочу продемонстрировать фактическое использование одного из них.

Предоставление возможности динамического выбора цвета

Стандартный диалог выбора цвета – это не украшение ради украшения. Сценарии могут передавать возвращаемую этим диалогом шестнадцатеричную строку цветов в уже знакомые нам параметры bg и fg настройки цветов виджетов. То есть параметры bg и fg принимают имя цвета (например, blue) и шестнадцатеричные строки со значения-

ми насыщенности цветов RGB, возвращаемые функцией `askcolor`, которые начинаются с `#` (например, `#8080ff` из последней строки вывода в предыдущем разделе).

Это добавляет новое измерение в модификацию графических интерфейсов на базе tkinter: вместо того чтобы жестко определять значения цветов в создаваемых интерфейсах, можно создать кнопку, выводящую диалог выбора цвета, с помощью которой пользователи смогут осуществлять настройку цветов на лету. Нужно просто передать строку цвета методу `config` в обработчиках событий, как показано в примере 8.11.

Пример 8.11. PP4E\Gui\Tour\setcolor.py

```
from tkinter import *
from tkinter.colorchooser 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 окно, изображенное на рис. 8.16 (фон его кнопки зеленоватый, и вам придется поверить мне на слово). Нажатие кнопки выводит диалог выбора цвета, который мы видели выше. Цвет, выбранный в этом окне, становится цветом фона этой кнопки после нажатия кнопки `OK` в диалоге.

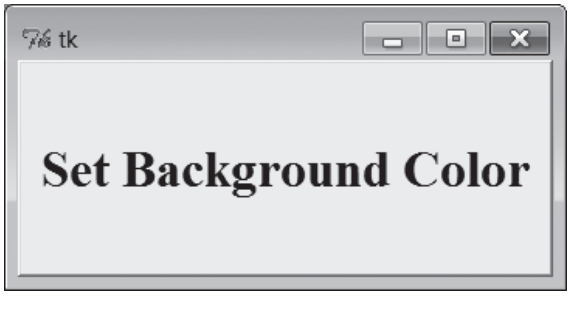


Рис. 8.16. *Главное окно setcolor*

Строки со значениями цвета также выводятся в поток `stdout` (окно консоли). Запустите этот сценарий на своем компьютере и поэкспериментируйте с возможными настройками цветов:

```
C:\...\PP4E\Gui\Tour> python setcolor.py
#0080c0
#408080
#77d5df
```

Другие стандартные диалоги

Мы уже видели большую часть стандартных диалогов, и мы будем пользоваться ими в примерах на протяжении оставшейся части книги. Если нужны дополнительные сведения о других имеющихся диалогах и параметрах, обращайтесь к другой документации по библиотеке tkinter или просмотрите исходные тексты модулей, используемых в начале модуля `dialogTable`, представленного в примере 8.8, – все они являются обычными файлами с программным кодом на языке Python, установленными на вашем компьютере в подкаталоге *tkinter* стандартной библиотеки Python (например, в каталоге *C:\Python31\Lib*, в Windows). И сохраните этот пример с демонстрационной панелью на будущее – мы снова воспользуемся им позднее, когда встретимся с другими виджетами, похожими на кнопки.

Модуль диалогов в старом стиле

В более старом программном коде на языке Python можно иногда увидеть диалоги, реализованные с использованием стандартного модуля `dialog`. Сейчас он несколько устарел и использует внешний вид, характерный для X Window, но на случай, если вам придется встретить такой программный код при сопровождении программ на языке Python, пример 8.12 может дать представление об этом интерфейсе.

Пример 8.12. PP4E\Gui\Tour\dlg-old.py

```
from tkinter import *
from tkinter.dialog import Dialog

class OldDialogDemo(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        Pack.config(self)              # то же, что и 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()
```

Если передать функции `Dialog` кортеж с метками для кнопок и текст сообщения, она вернет индекс нажатой кнопки (самая левая кнопка имеет индекс ноль). Окна `Dialog` являются модальными: доступ к остальным окнам приложения блокируется, пока `Dialog` ожидает ответа пользователя. При нажатии кнопки `Pop2` в главном окне этого сценария выводится второй диалог, как показано на рис. 8.17.

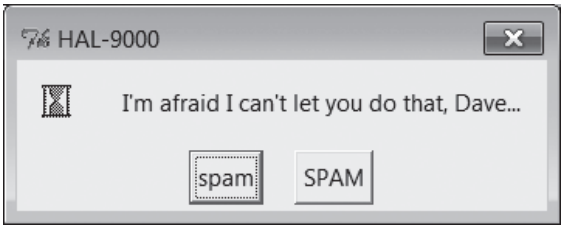


Рис. 8.17. *Диалог в старом стиле*

Сценарий был запущен в Windows, и, как видите, этот диалог нисколько не похож на то, что можно было бы ожидать на этой платформе. При вызове на любой платформе этот диалог имеет внешний вид, принятый в X Window. Из-за внешнего вида диалогов, воспроизводимых модулем `dialog` и повышенной сложности его использования лучше использовать стандартные диалоги, продемонстрированные в предыдущем разделе.

Пользовательские диалоги

Все встреченные нами до сих пор диалоги имеют стандартный внешний вид и способы взаимодействия. Для многих задач этого достаточно, но иногда требуется нечто более специфическое. Например, формы, требующие заполнения нескольких полей ввода (например, имя, возраст и размер обуви), не поддерживаются непосредственно библиотекой стандартных диалогов. Можно было бы поочередно выводить диалоги для ввода каждого значения, но такой интерфейс нельзя назвать дружественным.

Пользовательские диалоги поддерживают произвольные интерфейсы, но работать с ними сложнее. Впрочем, многого для этого не требуется: создать окно, такое как `Toplevel`, с прикрепленными виджетами и добавить обработчик события, который соберет данные, введенные пользователем (если они есть), и закроет окно. Чтобы сделать такой диалог модальным, необходимо передать окну фокус ввода, сделать другие

окна неактивными и ожидать события. Реализация такого диалога демонстрируется в примере 8.13.

Пример 8.13. PP4E\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 виджеты
    Button(win, text='OK', command=win.destroy).pack() # установить обработчик
    if makemodal:
        win.focus_set()    # принять фокус ввода,
        win.grab_set()     # запретить доступ к др. окнам, пока открыт диалог
        win.wait_window()  # ждать, пока win не будет уничтожен
    print('dialog exit')   # иначе – сразу вернуть управление

root = Tk()
Button(root, text='popup', command=dialog).pack()
root.mainloop()
```

Этот сценарий создает модальное или немодальное окно, в зависимости от значения глобальной переменной `makemodal`. Если запустить его без аргументов командной строки, выбирается немодальный стиль, как показано на рис. 8.18.

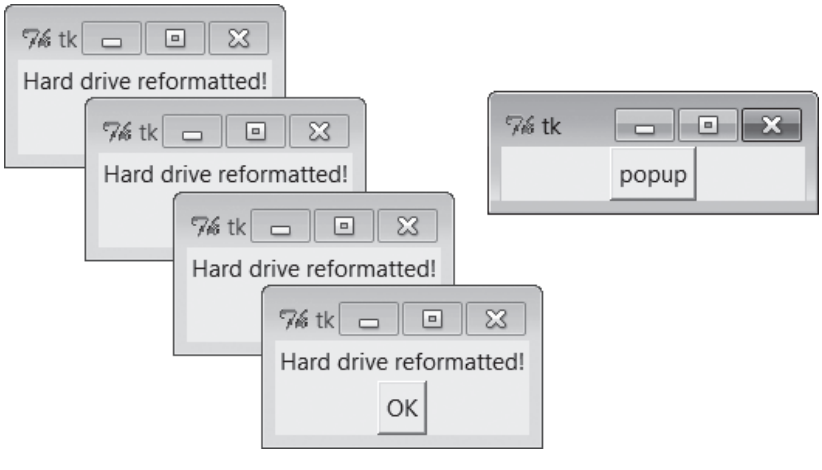


Рис. 8.18. *Немодальные пользовательские диалоги в действии*

Окно справа сверху – это корневое окно. При нажатии в нем кнопки `popup` создается новое диалоговое окно. Поскольку в этом режиме диалоги являются немодальными, корневое окно сохраняет активность после вывода диалога. Немодальные диалоги не блокируют другие окна,

поэтому кнопку в корневом окне можно нажать несколько раз и создать столько копий диалога, сколько поместится на экране. Любые из этих окон можно закрыть щелчком на их кнопках OK, при этом остальные окна останутся на экране.

Создание модальных пользовательских диалогов

Если запустить сценарий, передав ему аргумент в командной строке (например, `python dlg-custom.py 1`), окно диалога будет сделано модальным. Так как модальные диалоги сосредоточивают на себе все внимание интерфейса, главное окно становится недоступным, пока не будет закрыто окно диалога – пока диалог открыт, нельзя даже щелкнуть на корневом окне, чтобы активизировать его. Поэтому невозможно создать на экране больше одного всплывающего окна, как показано на рис. 8.19.

Рис. 8.19. *Модальный пользовательский диалог в действии*

Фактически функция `dialog` в этом сценарии не возвращает управление, пока диалог в левой части не будет закрыт нажатием кнопки OK. В результате модальные диалоги накладывают на модель программирования, в других случаях управляемую событиями, модель вызова функций – введенные пользователем данные можно обрабатывать сразу, а не в обработчике события, который вызывается в какой-то неопределенный момент времени в будущем.

Однако навязывание такой линейной логики управления в графическом интерфейсе требует некоторой дополнительной работы. Секрет блокирования других окон и ожидания ответа сводится к трем строкам программного кода, являющимся общим шаблоном для большинства пользовательских модальных диалогов:

`win.focus_set()`

Передает окну фокус ввода приложения, как если бы оно было активизировано щелчком мыши. У этого метода есть также синоним, `focus`, и часто фокус ввода устанавливается не на все окно, а на виджет в нем, позволяющий вводить данные (например, `Entry`).

`win.grab_set()`

Блокирует доступ ко всем другим окнам приложения, пока не будет закрыто данное окно. В это время пользователь не может взаимодействовать с другими окнами программы.

```
win.wait_window()
```

Приостанавливает вызвавшую программу, пока не будет уничтожен виджет `win`, но при этом главный цикл обработки событий (`mainloop`) остается активным. Это означает, что графический интерфейс в целом остается активным во время ожидания. Например, его окна перерисовываются при скрытии под другими окнами или открытии. Когда окно закрывается вызовом метода `destroy`, оно удаляется с экрана, блокировка приложения автоматически снимается и происходит возврат из данного метода.

Так как сценарий ждет события закрытия окна, он должен предоставить обработчик события, уничтожающий окно в ответ на взаимодействие с виджетами в диалоговом окне (единственном, которое активно). Диалог в этом примере является простым информационным диалогом, поэтому его кнопка `OK` вызывает метод `destroy` окна. В диалогах для ввода данных можно установить обработчик события нажатия клавиши `Enter`, который извлечет данные, введенные в элемент `Entry`, и после этого вызовет `destroy` (как будет показано далее в этой главе).

Другие способы реализации модальности

Модальные диалоги обычно реализуются путем создания нового всплывающего окна и ожидания в нем 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 `deiconify` и `withdraw` окна верхнего уровня (подробности смотрите в сценариях раздела главы 9). С учетом того, что в настоящее время скорость выполнения такова, что создание окон происходит практически мгновенно, такой способ встречается значительно реже, чем создание окон с нуля и уничтожение 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 9. В этой схеме обработчик события долгоживущего диалогового окна 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` виджета. Метод `quit` прекращает выполнение функции `mainloop` и потому обычно завершает выполнение программы с графическим интерфейсом. Но если был произведен рекурсивный вызов `mainloop`, метод `quit` просто завершит его. Благодаря этому модальные диалоги можно реализовать без обраще-

ния к методу ожидания. Так, сценарий в примере 8.14 работает аналогично dlg-custom в модальном режиме.

Пример 8.14. PP4E\Gui\Tour\dlg-recursive.py

```
from tkinter import *

def dialog():
    win = Toplevel()                                  # создать новое окно
    Label(win, text='Hard drive reformatted!').pack() # добавить виджеты
    Button(win, text='OK', command=win.quit).pack()   # установить обр-к quit
    win.protocol('WM_DELETE_WINDOW', win.quit)        # завершить и при
                                                      # закрытии окна!

    win.focus_set()    # принять фокус ввода,
    win.grab_set()     # запретить доступ к др. окнам, пока открыт диалог
    win.mainloop()     # и запустить вложенный цикл обр. событий для ожидания
    win.destroy()
    print('dialog exit')

root = Tk()
Button(root, text='popup', command=dialog).pack()
root.mainloop()
```

Выбирая этот путь, нужно вместо метода destroy вызывать в обработчиках событий метод quit (destroy не завершает функцию mainloop) и обеспечить вызов quit кнопкой закрытия окна с помощью метода protocol (иначе не будет завершаться рекурсивный вызов 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 диалоги в виде форм с метками и полями ввода, мы увидим далее в этой главе, познакомившись с элементом Entry, и еще раз – при изучении менеджера grid в главе 9. Другие примеры пользовательских диалогов можно найти в демонстрационных приложениях ShellGui (глава 10), PyMailGui (глава 14), PyCalc (глава 19) и немодальном *form.py* (глава 12). А сейчас мы перейдем к более глубокому изучению событий, что несомненно пригодится на следующих этапах нашего турне.

Привязка событий

В предыдущей главе мы познакомились с методом bind виджетов, который использовался для перехвата нажатий кнопок. Так как метод bind часто используется вместе с другими виджетами (например, для перехвата нажатия клавиши Enter в полях ввода), остановимся на нем здесь в начале нашего обзора. Пример 8.15 иллюстрирует другие протоколы событий для метода bind.

Пример 8.15. PP4E\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):
        if not attr.startswith('__'):
            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:', end=' ')
    showPosEvent(event)

def onRightClick(event):
    print('Got right mouse button click:', end=' ')
    showPosEvent(event)

def onMiddleClick(event):
    print('Got middle mouse button click:', end=' ')
    showPosEvent(event)
    showAllEvent(event)

def onLeftDrag(event):
    print('Got left mouse button drag:', end=' ')
    showPosEvent(event)

def onDoubleLeftClick(event):
    print('Got double left mouse click', end=' ')
    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, большой шрифт
widget.config(height=5, width=20)            # начальн. размер: строк,символов
widget.pack(expand=YES, fill=BOTH)
widget.bind('<Button-1>',  onLeftClick)      # щелчок мышью
```

```
widget.bind('<Button-3>',  onRightClick)
widget.bind('<Button-2>',  onMiddleClick)    # средняя = обе на некот. мышах
widget.bind('<Double-1>',  onDoubleLeftClick)# двойной щелчок левой кнопкой
widget.bind('<B1-Motion>', onLeftDrag)       # щелчок левой кнопкой и перемещ.
widget.bind('<KeyPress>',  onKeyPress)       # нажатие любой клавиши на клав.
widget.bind('<Up>',        onArrowKey)       # нажатие клавиши со стрелкой
widget.bind('<Return>',    onReturnKey)      # return/enter key pressed
widget.focus()                               # или привязать нажатие клавиши
                                             # к tkroot

tkroot.title('Click Me')
tkroot.mainloop()
```

Этот файл состоит в основном из функций обработчиков событий, вызываемых при возникновении связанных событий. Как было показано в главе 7, обработчики данного типа получают в качестве аргумента объект события, содержащий сведения о сгенерированном событии. Технически этот аргумент является экземпляром класса `Event` из библиотеки tkinter, и содержащиеся в нем подробности представлены атрибутами. Большинство обработчиков просто выводят информацию о событиях, извлекая значения из их атрибутов.

Если запустить этот сценарий, он создаст окно, изображенное на рис. 8.20. Главное его назначение – служить областью для запуска событий щелчков мышью и нажатия клавиш.

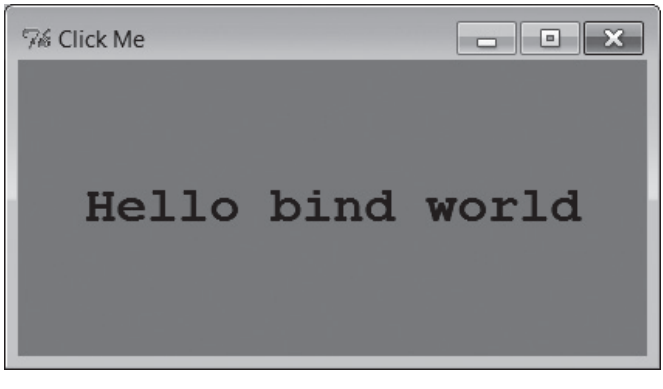


***Рис. 8.20.** Окно сценария bind для щелчков мышью*

Черно-белое издание, которое вы держите в руках, не позволяет оценить этот сценарий. При запуске вживую 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 встречали в главе 7 сценарий, который с помощью метода `bind` виджета и имен событий `<Button-1>` и `<Double-1>` перехватывал одиночные и двойные щелчки левой кнопкой

мыши. Данный сценарий демонстрирует другие виды событий, часто перехватываемые с помощью метода `bind`:

`<KeyPress>`

Чтобы перехватывать нажатия одиночных клавиш на клавиатуре, можно зарегистрировать обработчик для события с идентификатором `<KeyPress>` – это более низкоуровневый способ ввода данных в программах с графическим интерфейсом, чем использование виджета `Entry`, о котором рассказывается в следующем разделе. Нажатая клавиша возвращается в виде кода ASCII в объекте события, передаваемом обработчику события (`event.char`). Другие атрибуты в структуре события позволяют идентифицировать нажатую клавишу на еще более низком уровне. Нажатия клавиш можно перехватывать виджетом корневого окна верхнего уровня или виджетом, которому передан фокус ввода 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 текущие координаты X/Y указателя мыши (`event.x`, `event.y`). Эту информацию можно использовать для организации перемещения объектов, перетаскивания, рисования на уровне пикселей и так далее (смотрите демонстрационный пример программы PyDraw в главе 11).

`<Button-3>`, `<Button-2>`

Этот сценарий перехватывает также щелчки правой и средней кнопками мыши (называемыми также кнопками 3 и 2). Воспроизвести щелчок средней кнопкой 2 с помощью двухкнопочной мыши можно, щелкнув одновременно обеими кнопками, – если этот прием не действует, проверьте настройки мыши в интерфейсе свойств (`Панель управления` (`Control Panel`) в Windows)[1].

`<Return>`, `<Up>`

Чтобы перехватывать нажатия более специфических клавиш, в данном сценарии зарегистрированы обработчики событий нажатия клавиш `Return`/`Enter` и «стрелки вверх». В противном случае эти события были бы отправлены универсальному обработчику события `<KeyPress>` и потребовали бы дополнительного анализа события.

Ниже показано, что попадает в поток вывода `stdout` после щелчка левой кнопкой, правой кнопкой, левой кнопкой и перетаскивания, нескольких нажатий клавиш, нажатия клавиш `Enter` и «стрелки вверх» и на-

[1] Видимо, у автора установлены какие-то дополнительные драйверы от производителя мыши. В стандартной поставке Windows такая настройка отсутствует. – *Прим. ред.*

конец, двойного щелчка левой кнопкой для завершения. При нажатии левой кнопки мыши и перемещении курсора по экрану возникает множество сообщений с информацией о событии перетаскивания – одно сообщение выводится для каждого движения при перетаскивании (и для каждого производится вызов обработчика на языке Python):

```
C:\...\PP4E\Gui\Tour> python bind.py
Got left mouse button click: Widget=.25763696 X=376 Y=53
Got right mouse button click: Widget=.25763696 X=36 Y=60
Got left mouse button click: Widget=.25763696 X=144 Y=43
Got left mouse button drag: Widget=.25763696 X=144 Y=45
Got left mouse button drag: Widget=.25763696 X=144 Y=47
Got left mouse button drag: Widget=.25763696 X=145 Y=50
Got left mouse button drag: Widget=.25763696 X=146 Y=51
Got left mouse button drag: Widget=.25763696 X=149 Y=53
Got key press: s
Got key press: p
Got key press: a
Got key press: m
Got key press: 1
Got key press: -
Got key press: 2
Got key press: .
Got return key press
Got up arrow key press
Got left mouse button click: Widget=.25763696 X=300 Y=68
Got double left mouse click Widget=.25763696 X=300 Y=68
```

Для событий, связанных с мышью, обработчики выводят координаты X и Y указателя мыши, которые передаются в объекте события. Обычно координаты измеряются в пикселях от верхнего левого угла (0, 0), относительно того виджета, на котором произведен щелчок. Ниже показано, что выводится для щелчка левой кнопкой, средней кнопкой и двойного щелчка левой. Обратите внимание, что обработчик щелчка средней кнопкой выводит свой аргумент целиком – все атрибуты объекта `Event` (исключая внутренние атрибуты с именами, начинающимися с двух символов подчеркивания «__», в число которых входит атрибут `__doc__` и методы перегрузки операторов, унаследованные от суперкласса `object`, подразумеваемого в Python 3.X по умолчанию). Различные типы событий устанавливают различные атрибуты. Например, нажатие большинства клавиш записывает некоторое значение в атрибут `char`:

```
C:\...\PP4E\Gui\Tour> python bind.py
Got left mouse button click: Widget=.25632624 X=6 Y=6
Got middle mouse button click: Widget=.25632624 X=212 Y=95
<tkinter.Event object at 0x018CA210>
char => ??
delta => 0
height => ??
keycode => ??
```

```
keysym => ??
keysym_num => ??
num => 2
send_event => False
serial => 17
state => 0
time => 549707945
type => 4
widget => .25632624
width => ??
x => 212
x_root => 311
y => 95
y_root => 221
Got left mouse button click: Widget=.25632624 X=400 Y=183
Got double left mouse click Widget=.25632624 X=400 Y=183
```

Другие события, доступные с помощью метода bind

Помимо событий, которые были проиллюстрированы в данном примере, сценарий, использующий библиотеку tkinter, может зарегистрировать обработчики других видов связываемых событий. Например:

- `<ButtonRelease>` генерируется при отпускании кнопки мыши (событие `<ButtonPress>` генерируется, когда кнопка нажимается).
- `<Motion>` генерируется при перемещении указателя мыши.
- Обработчики `<Enter>` и `<Leave>` генерируются в момент входа и выхода указателя мыши из области окна (полезно для автоматического выделения виджета).
- `<Configure>` генерируется при изменении размеров окна, его положения и так далее (например, новые размеры окна содержатся в атрибутах `width` и `height` объекта события). Мы будем использовать это событие для масштабирования содержимого окна при изменении его размеров в примере PyClock, в главе 11.
- `<Destroy>` генерируется при уничтожении виджета окна (и отличается от механизма `protocol` менеджера окон, реализованного для кнопки закрытия). Поскольку это событие имеет непосредственное отношение к методам `quit` и `destroy` виджетов, я расскажу о нем более подробно далее в этом разделе.
- `<FocusIn>` и `<FocusOut>` генерируются, когда виджет получает или теряет фокус ввода.
- `<Map>` и `<Unmap>` генерируются, когда окно сворачивается в значок и восстанавливается.
- `<Escape>`, `<BackSpace>` и `<Tab>` генерируются при нажатии других специальных клавиш.

- <Down>, <Left> и <Right> генерируются при нажатии 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 указателя мыши при нажатой левой кнопке, а <KeyPress-a> генерируется только при нажатии 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 формы имен чувствительны к регистру символов: пишите <Key-Escape>, а не <KEY-ESCAPE>.
- Имеется возможность 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 источникам информации по Tk и tkinter. Кроме того, изменяя настройки в сценарии и запуская его заново, также можно выяснить некоторые особенности поведения событий – в конце концов, это Python.

Подробнее о событии <Destroy> и методах quit и destroy

Прежде чем двинуться дальше, необходимо сказать несколько слов о событии <Destroy> (регистр символов в имени которого имеет значение): это событие генерируется, когда выполняется операция уничтожения виджета, будь то вызов метода из сценария или операция закрытия окна пользователем, включая завершение программы. Если привязать обработчик этого события к окну, он будет вызываться по одному разу для каждого виджета в окне – атрибут widget объекта события, передаваемого обработчику в виде аргумента, будет ссылаться на уничтожаемый виджет, и вы можете использовать эту особенность, чтобы определить момент уничтожения какого-то определенного виджета. Если же привязать этот обработчик к какому-то определенному виджету, он будет вызываться только при уничтожении этого виджета.

Важно знать, что в момент возбуждения события виджет находится в «полумертвом» состоянии (в терминологии библиотеки Tk) – он по-прежнему существует, но большинство операций над ним будут терпеть неудачу. По этой причине событие <Destroy> вообще не может использоваться для выполнения операций с графическим интерфейсом – например, попытки проверки признака изменения состояния виджета или извлечения его содержимого в обработчике события <Destroy> будут возбуждать исключения. Кроме того, в этом обработчике нельзя отме-

нить уничтожение виджетов и обеспечить продолжение работы графического интерфейса. Если вам потребуется перехватывать и проверять или подавлять операцию закрытия окна после щелчка пользователем на кнопке X окна, используйте событие WM_DELETE_WINDOW в методе protocol высокого уровня, как было описано выше в этой главе.

Кроме того, вы должны знать, что вызов метода quit виджетов не возбуждает никаких событий <Destroy>, а наличие любых зарегистрированных обработчиков события <Destroy> в программах, выполняемых под управлением Python 3.X, вообще приводит к фатальной ошибке при завершении программы. Вследствие этого программы, привязывающие обработчики этого события для выполнения заключительных действий, не связанных с графическим интерфейсом, должны обычно вызывать метод destroy вместо quit и надеяться, что программа завершится вместе с уничтожением последнего или единственного корневого окна Tk (созданного явно или по умолчанию), как описывалось выше. Это обстоятельство препятствует использованию метода quit для немедленного завершения программы, хотя у вас всегда остается последнее средство – функция sys.exit.

Сценарий может также выполнять заключительные операции в программном коде, следующем за вызовом функции mainloop, но к этому моменту графический интерфейс уже будет полностью уничтожен, и данный программный код не может быть привязан к какому-то конкретному виджету. Мы еще будем говорить об этом событии, когда будем изучать программу PyEdit в главе 11 – мы найдем это событие непригодным для проверки наличия изменений в тексте, определяющих необходимость его сохранения.

Виджеты Message и Entry

Виджеты Message и Entry позволяют отображать и вводить простой текст. Оба они, в сущности, являются функциональными подмножествами виджета Text, с которым мы познакомимся позднее, – Text может делать все то, что могут Message и Entry, при этом обратное утверждение неверно.

Message

Виджет Message служит всего лишь местом для отображения текста. Хотя с помощью стандартного диалога showinfo, с которым мы встречались ранее, выводить всплывающие сообщения, вероятно, удобнее, тем не менее виджет Message автоматически и гибко разбивает длинные строки и может встраиваться внутрь элементов-контейнеров, когда нужно вывести на экране какой-либо текст, доступный только для чтения. Кроме того, этот виджет обладает более чем десятком параметров настройки, позволяющих изменять его внешний вид. Пример 8.16 и рис. 8.21 иллюстрируют основы применения Message и демонстрируют, как этот виджет реагирует на растягивание по горизонтали с при-

менением параметров `fill` и `expand`. Дополнительные сведения об изменении размеров виджетов вы найдете в главе 7, а сведения о других поддерживаемых параметрах ищите в справочниках по Tk или tkinter.

Пример 8.16. PP4E\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(fill=X, expand=YES)
mainloop()
```

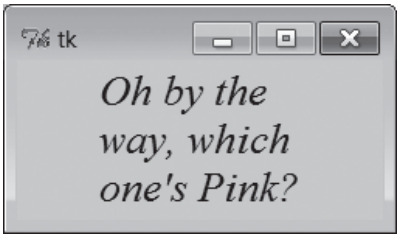


***Рис. 8.21.** Виджет Message в действии*

Entry

Виджет `Entry` служит простым полем ввода одной строки текста. Обычно он используется для реализации полей ввода в диалогах, имеющих вид форм, и всюду, где пользователь должен ввести значение в поле. Виджет `Entry` также поддерживает более сложные понятия, такие как прокрутка, привязка клавиш для редактирования и выделение текста, при этом он очень прост в использовании. Сценарий в примере 8.17 создает окно для ввода, изображенное на рис. 8.22.

Пример 8.17. PP4E\Gui\tour\entry1.py

```
from tkinter import *
from quitter import Quitter

def fetch():
    print('Input => "%s"' % ent.get())           # извлечь текст

root = Tk()
ent = Entry(root)
ent.insert(0, 'Type words here')                 # записать текст
ent.pack(side=TOP, fill=X)                       # растянуть по горизонтали

ent.focus()                                      # избавить от необходимости
                                                 # выполнять щелчок мышью
ent.bind('<Return>', (lambda event: fetch()))    # по нажатию клавиши Enter
btn = Button(root, text='Fetch', command=fetch)  # и по щелчку на кнопке
btn.pack(side=LEFT)
```

```
Quitter(root).pack(side=RIGHT)
root.mainloop()
```

***Рис. 8.22.** Сценарий entry1 в действии*

Если запустить сценарий `entry1`, он заполнит поле ввода в этом интерфейсе текстом «Type words here» вызовом метода `insert` виджета. Поскольку щелчок на кнопке Fetch и нажатие клавиши Enter запускают в сценарии функцию обратного вызова `fetch`, оба эти события извлекут из поля ввода текущий текст с помощью метода `get` виджета и выведут его:

```
C:\...\PP4E\Gui\Tour> python entry1.py
Input => "Type words here"
Input => "Have a cigar"
```

Мы уже встречались выше с событием `<Return>`, когда знакомились с методом `bind` – в отличие от событий нажатий на кнопки, эти низкоуровневые обработчики получают в качестве аргумента объект события, поэтому, чтобы игнорировать его, в сценарии использовано обертывающее `lambda`-выражение. Кроме того, поле ввода в этом сценарии компонуется с параметром `fill=X`, чтобы оно растягивалось по горизонтали вместе с окном (попробуйте сами), и вызывается метод `focus` виджета, чтобы автоматически передать фокус в поле ввода при появлении окна. Благодаря передаче фокуса вручную пользователю не нужно щелкать на поле, чтобы начать ввод данных. Наша умная кнопка Quit, которую мы реализовали ранее, также прикрепляется к интерфейсу (она выводит диалог с просьбой подтвердить завершение приложения).

Программирование виджетов Entry

Вообще говоря, значения, вводимые в виджеты `Entry` и отображаемые ими, могут быть записаны или получены с помощью связанных объектов «переменных» (описываемых далее в этой главе) или с помощью следующих методов виджета `Entry`:

```
ent.insert(0, 'some text')      # запись значения
value = ent.get()               # извлечение значения (строки)
```

Первый параметр метода `insert` определяет позицию в строке, начиная с которой должен быть введен текст. Здесь «0» означает ввод в начало строки, поскольку смещения начинают отсчитываться с нуля, а целое число `0` и строка `'0'` означают одно и то же (аргументы методов в библиотеке tkinter всегда при необходимости преобразуются в строки). Если виджет `Entry` уже содержит текст, то обычно требуется удалить

его содержимое перед записью 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 записать значение
```

Имя END здесь является предопределенной константой tkinter, обозначающей конец содержимого виджета – она снова встретится нам в главе 9 при изучении полномасштабного и многострочного виджета Text (более мощного собрата Entry). Поскольку после удаления виджет не будет ничего содержать, предыдущая последовательность инструкций эквивалентна следующей:

```
ent.delete('0', END)          # удалить текст с начала до конца
ent.insert(END, 'some text')  # добавить в конец пустой строки текста
```

В любом случае, если сначала не удалить текст, новый текст просто будет добавлен к нему. Если вам интересно увидеть, как это происходит, измените функцию fetch, как показано ниже, и при каждом щелчке кнопкой или нажатии клавиши в начало и в 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 state='disabled' виджета Entry, делающим его доступным только для чтения, а также параметром show='*', заставляющим его выводить каждый символ как * (полезно для организации ввода паролей). Поэкспериментируйте с этим сценарием, изменяя и запуская его. Виджет Entry поддерживает и другие параметры, которые мы здесь также пропустим; дополнительные сведения ищите в последующих примерах и других источниках.

Компоновка элементов ввода в формах

Как уже отмечалось, виджеты Entry часто применяются в качестве полей ввода при реализации форм. Мы часто будем создавать такие формы в этой книге. Простую иллюстрацию такого применения дает пример 8.18, в котором несколько меток, полей ввода и фреймов объединены в форму для ввода нескольких значений, изображенную на рис. 8.23.

Пример 8.18. PP4E\Gui\Tour\entry2.py

```
"""
непосредственное использование виджетов Entry и размещение их по рядам с метками
фиксированной ширины: такой способ компоновки, а также использование менеджера
grid обеспечивают наилучшее представление для форм
"""
```

```
from tkinter import *
from quitter import Quitter
fields = 'Name', 'Job', 'Pay'

def fetch(entries):
    for entry in entries:
        print('Input => "%s"' % entry.get())     # извлечь 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)               # прикрепить к верхнему краю
        lab.pack(side=LEFT)
        ent.pack(side=RIGHT, expand=YES, fill=X) # растянуть по горизонтали
        entries.append(ent)
    return entries

if __name__ == '__main__':
    root = Tk()
    ents = makeform(root, fields)
    root.bind('<Return>', (lambda event: fetch(ents)))
    Button(root, text='Fetch',
                 command = (lambda: fetch(ents))).pack(side=LEFT)
    Quitter(root).pack(side=RIGHT)
    root.mainloop()
```

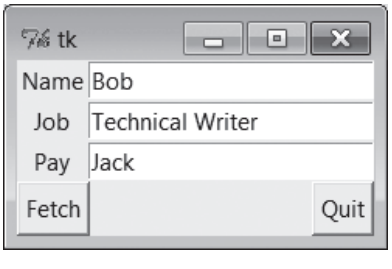


Рис. 8.23. *Внешний вид форм entry2 (и entry3)*

Полями ввода здесь служат простые виджеты `Entry`. Сценарий создает список виджетов, с помощью которого потом будут извлекаться их значения. При каждом нажатии кнопки Fetch текущие значения извлекаются из всех полей ввода и выводятся в стандартный поток вывода:

```
C:\...\PP4E\Gui\Tour> python entry2.py
Input => "Bob"
Input => "Technical Writer"
Input => "Jack"
```

Тот же результат дает нажатие клавиши Enter, когда окно обладает фокусом ввода, – 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 виджетов. В данном сценарии каждый ряд метка/поле ввода конструируется как новый фрейм `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 фрейм, его содержимое изолируется от других операций компоновки, производимых в этом окне. Кроме того, этот сценарий разрешает увеличение горизонтального размера при изменении размеров окна только для полей ввода, как показано на рис. 8.24.

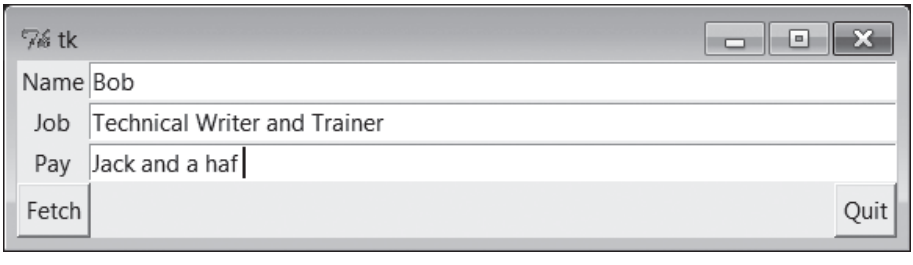


Рис. 8.24. *Возможность растягивания полей ввода в сценариях entry2 (и entry3) в действии*

Снова о создании модальных окон

Далее мы увидим, как создавать аналогичные структуры форм с помощью менеджера компоновки `grid`, где вместо фреймов размещение виджетов выполняется по номерам рядов и столбцов. Но сейчас, реализовав структуру формы, посмотрим, как применять технологию создания модальных диалогов к более сложным формам.

Сценарий в примере 8.19, используя функции `makeform` и `fetch` из предыдущего примера, создает форму и выводит 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 уничтожения окна. Как мы уже знаем, метод `wait_window` влечет приостановку программы, пока окно не будет закрыто.

Пример 8.19. PP4E\Gui\Tour\entry2-modal.py

```
# создает модальный диалог с формой;
# данные должны извлекаться до уничтожения окна с полями ввода

from tkinter import *
from entry2 import makeform, fetch, fields
```

```
def show(entries, popup):
    fetch(entries)              # извлечь данные перед уничтожением окна!
    popup.destroy()             # если инструкции поменять местами, сценарий
                                # будет возбуждать исключение
def ask():
    popup = Toplevel()          # отобразить форму в виде модального диалога
    ents = makeform(popup, fields)
    Button(popup, text='OK', command=(lambda: show(ents, popup))).pack()
    popup.grab_set()
    popup.focus_set()
    popup.wait_window()         # ждать закрытия окна

root = Tk()
Button(root, text='Dialog', command=ask).pack()
root.mainloop()
```

Если нажать кнопку в главном окне, сценарий создаст окно диалога с формой, блокирующее остальное приложение, изображенное на рис. 8.25.

В реализации этого модального диалога таится малозаметная опасность: поскольку он извлекает данные, вводимые пользователем, из виджетов `Entry`, встроенных во всплывающее окно, эти данные необходимо получить *прежде* чем окно будет уничтожено в обработчике события нажатия кнопки `OK`. Оказывается, что вызов `destroy` действительно уничтожает все виджеты окна – попытка получить значение из уничтоженного виджета `Entry` не только не действует, но и порождает исключение с выводом трассировочной информации и сообщения об ошибке в окне консоли – попробуйте изменить порядок команд в функции `show`, и вы убедитесь в этом сами.

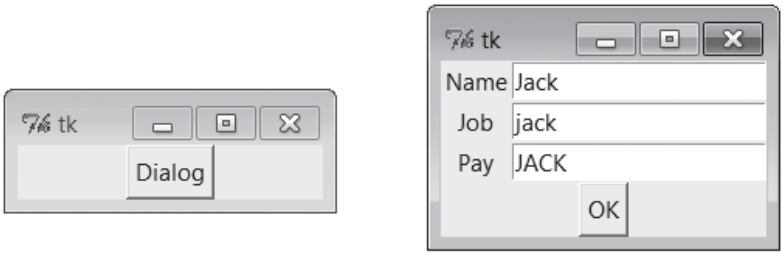


***Рис. 8.25.** Окна, создаваемые сценарием entry2-modal (и entry3-modal)*

Чтобы избежать этой проблемы, нужно следить за тем, чтобы выборка значений осуществлялась перед уничтожением, или использовать переменные tkinter, являющиеся предметом обсуждения следующего раздела.

«Переменные» tkinter и альтернативные способы компоновки форм

Виджеты Entry (наряду с другими) поддерживают понятие ассоциированной переменной – изменение значения ассоциированной переменной изменяет текст, отображаемый виджетом Entry, а изменение текста в Entry изменяет значение переменной. Однако это не обычные переменные Python. Переменные, связанные с виджетами, являются экземплярами классов переменных в библиотеке tkinter. Эти классы носят названия StringVar, IntVar, DoubleVar и BooleanVar. Выбор того или иного класса зависит от контекста, в котором он должен использоваться. Например, можно связать с полем Entry экземпляр класса StringVar, как показано в примере 8.20.

Пример 8.20. PP4E\Gui\Tour\entry3.py

```
"""
использует переменные StringVar
компоновка по колонкам: вертикальные координаты виджетов могут не совпадать
(смотрите entry2)
"""

from tkinter import *
from quitter import Quitter
fields = 'Name', 'Job', 'Pay'

def fetch(variables):
    for variable in variables:
        print('Input => "%s"' % variable.get())  # извлечь 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)    # растягивать 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)               # растягивать 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: fetch(vars))).pack(side=LEFT)
    Quitter(root).pack(side=RIGHT)
    root.bind('<Return>', (lambda event: fetch(vars)))
    root.mainloop()
```

За исключением того обстоятельства, что поля ввода инициализируются строкой `'enter here'`, этот сценарий создает окно, практически идентичное по внешнему виду и функциям тому, которое создает сценарий `entry2` (рис. 8.23 и 8.24). Для наглядности виджеты в окне компонуются другим способом – как фрейм с двумя вложенными фреймами, образующими левую и правую колонки в области формы, – но конечный результат при отображении на экран оказывается тем же самым (на некоторых платформах, по крайней мере: смотрите примечание в конце этого раздела, где описывается, почему компоновка на основе рядов обычно бывает предпочтительнее).

Главное, на что здесь нужно обратить внимание, это использование переменных `StringVar`. Вместо списка виджетов `Entry`, из которого извлекаются введенные значения, эта версия хранит список объектов `StringVar`, которые ассоциируются с виджетами `Entry` следующим способом:

```
ent = Entry(rite)
var = StringVar()
ent.config(textvariable=var)   # связать поле с переменной
```

После того как переменные будут связаны, операции изменения и получения значения переменной

```
var.set('text here')
value = var.get()
```

действительно будут изменять и получать значение соответствующего поля ввода на экране.[1] Метод `get` объекта переменной возвращает строку для `StringVar`, целое число для `IntVar` и число с плавающей точкой для `DoubleVar`.

---

[1] Исторический анекдот: в устаревшей в данное время версии tkinter, поставлявшейся с Python 1.3, можно было также устанавливать и извлекать значения переменных, обращаясь к ним как к функциям с аргументами или без (например, `var(value)` и `var()`). В настоящее время вместо этого нужно вызывать методы `set` и `get` переменных. По неустановленным причинам функциональная форма вызова прекратила работать годы назад, хотя ее все еще можно встретить в старом программном коде на языке Python (и в первых изданиях, по крайней мере, одной книги по языку Python, выпущенной издательством O'Reilly). Если исправление, выполненное из эстетических соображений, приводит к неработоспособности существующего программного кода, можно ли его считать исправлением?

Конечно, как мы уже видели, можно легко изменять и извлекать текст непосредственно из полей Entry, без всяких дополнительных переменных. Зачем же утруждать себя обработкой объектов переменных? Во-первых, исчезает опасность попыток извлечения значений после уничтожения, о чем говорилось в предыдущем разделе. Поскольку объекты StringVar продолжают существовать после уничтожения виджетов Entry, к которым они привязаны, сохраняется возможность извлекать из них значения, когда модального диалога уже давно нет, как показано в примере 8.21.

Пример 8.21. PP4E\Gui\Tour\entry3-modal.py

```
# значения могут извлекаться из StringVar и после уничтожения виджета

from tkinter import *
from entry3 import makeform, fetch, fields

def show(variables, popup):
    popup.destroy()            # здесь порядок не имеет значения
    fetch(variables)           # переменные сохраняются после уничтожения окна

def ask():
    popup = Toplevel()         # отображение формы в модальном диалоге
    vars = makeform(popup, fields)
    Button(popup, text='OK', command=(lambda: show(vars, popup))).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 (представленная в примере 8.19 и на рис. 8.25), но теперь функция show уничтожает всплывающее окно до извлечения введенных данных из переменных StringVar в списке, созданном функцией makeform. Иными словами, переменные оказываются более надежными в некоторых контекстах, потому что они не являются частью действительного дерева виджетов. Например, они также часто используются с флажками, группами переключателей и ползунками, обеспечивая доступ к текущим значениям и связывая вместе несколько виджетов. Так уж совпало, что им посвящен следующий раздел.

В этом разделе мы использовали два способа компоновки форм: во фреймах *по рядам*, с метками фиксированной ширины (entry2), и во фреймах *по колонкам* (entry3). В главе 9 мы познакомимся с третьим способом: компоновкой с помощью менеджера grid. Из них наилучший результат на всех платформах обеспечивают компоновка по сетке и по рядам с метками фиксированной ширины, как в сценарии entry2.

Компоновка по колонкам, использованная в сценарии entry3, может использоваться только на платформах, где высота каждой метки в точности соответствует высоте каждого поля ввода. Поскольку эти виджеты напрямую никак не связаны, их вертикальные координаты могут не совпадать на некоторых платформах. Когда я попытался протестировать в системе Linux некоторые формы, замечательно выглядевшие в Windows XP, метки и соответствующие им поля ввода оказались на разной высоте.

Даже такое простое окно, которое воспроизводит сценарий entry3, при ближайшем рассмотрении выглядит несколько кривовато. На некоторых платформах оно только кажется похожим на окно, воспроизводимое сценарием entry2, из-за небольшого количества полей ввода и небольших размеров по умолчанию. В Windows 7 на моем нетбуке несовпадение меток и полей ввода по вертикали становится заметным после добавления 3–4 дополнительных полей ввода в кортеж полей в сценарии entry3.

Если переносимость имеет для вас важное значение, компонуйте свои формы либо с помощью фреймов по рядам и с метками фиксированной/максимальной ширины, как в сценарии entry2, либо с выравниванием виджетов по сетке. Дополнительные примеры таких форм мы увидим в следующей главе. А в главе 12 мы напишем свой инструмент конструирования форм, скрывающий тонкости их компоновки от клиента (включая пример клиента в главе 13).

Флажки, переключатели и ползунки

Этот раздел знакомит с тремя типами виджетов – Checkbutton («флажок», виджет для выбора нескольких вариантов одновременно), Radiobutton («переключатель», виджет для выбора единственного варианта из нескольких) и Scale («шкала», 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, представленный в примере 8.8, где определяются обработчики событий выбора виджетов (обработчики, вызывающие диалоги). Попутно мы воспользуемся только что рассмотренными переменными tkinter для получения значений состояния этих виджетов.

Флажки

Виджеты Checkbutton и Radiobutton предусматривают возможность ассоциирования с переменными tkinter: щелчок на виджете изменяет значение переменной, а изменение значения переменной изменяет со-

стояние виджета, к которому она привязана. В действительности переменные tkinter составляют функциональную основу этих графических элементов:

- Группа флажков `Checkbutton` реализует интерфейс с выбором нескольких вариантов путем присвоения каждому виджету (флажку) собственной переменной.
- Группа переключателей `Radiobutton` реализует модель выбора единственного из нескольких взаимоисключающих вариантов путем придания каждому виджету уникального значения и назначения одной и той же переменной tkinter.

У обоих типов виджетов есть параметры `command` и `variable`. Параметр `command` позволяет зарегистрировать обработчик, который вызывается, как только возникает событие щелчка на виджете, подобно обычным виджетам `Button`. Но передавая переменную tkinter в параметре `variable`, можно также в любой момент получать или изменять состояние виджета путем получения или изменения значения связанной с ним переменной.

Флажки tkinter несколько проще в обращении, поэтому с них и начнем. Пример 8.22 создает группу из пяти флажков, изображенную на рис. 8.26. Для большей пользы он также добавляет кнопку, с помощью которой выводится текущее состояние всех флажков, и прикрепляет экземпляр кнопки `Quitter`, которую мы создали в начале главы.

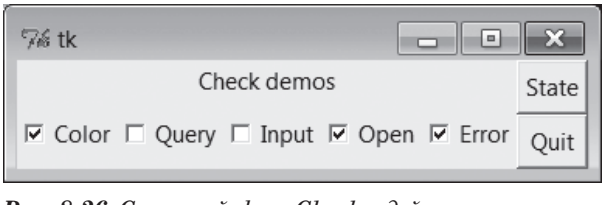


***Рис. 8.26.** Сценарий demoCheck в действии*

Пример 8.22. PP4E\Gui\Tour\demoCheck.py

```
"создает группу флажков, которые вызывают демонстрационные диалоги"

from tkinter import *                   # импортировать базовый набор виджетов
from dialogTable import demos           # импортировать готовые диалоги
from quitter import Quitter             # прикрепить к "себе" объект Quitter

class Demo(Frame):
    def __init__(self, parent=None, **options):
        Frame.__init__(self, parent, **options)
        self.pack()
        self.tools()
        Label(self, text="Check demos").pack()
```

```
            self.vars = []
            for key in demos:
                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(), end=' ')  # текущие значения флажков: 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()
```

С точки зрения программного кода, флажки похожи на обычные кнопки, они даже добавляются в контейнерный виджет. Однако функционально они несколько отличаются. Как можно видеть по рисунку (а лучше – запустив пример), флажок работает как переключатель: щелчок на нем изменяет его состояние из выключенного во включенное (из невыбранного 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, галочка исчезает, а его переменная `IntVar` получает значение `0`.

Чтобы смоделировать приложение, содержащее флажки, кнопка State в этом графическом интерфейсе запускает метод `report` в сценарии, который выводит текущие состояния всех пяти флажков в поток `stdout`. Ниже приводится вывод, полученный после нескольких щелчков:

```
C:\...\PP4E\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 `Checkbutton` посредством параметров `variable`, и при опросе они совпадают со значениями виджетов. В этом сценарии с каждым из флажков `Checkbutton` на экране ассоциирована переменная `IntVar`, поскольку это двоичные индикаторы, способные принимать значение `0` или `1`. Переменные `StringVar` тоже можно использовать, но при

этом их методы будут возвращать строки '0' или '1', а не целые числа, а их начальным состоянием будет пустая строка (а не целое число 0).

Параметр command этого виджета позволяет зарегистрировать обработчик, который будет вызываться при каждом щелчке на виджете. Для иллюстрации в качестве обработчика для каждого из флажков в этом сценарии зарегистрирован вызов демонстрации стандартного диалога: щелчок изменяет состояние переключателя, а кроме того, выводит один из знакомых диалогов.

Интересно, что вызвать метод report можно также в интерактивном сеансе. При работе в таком режиме виджеты отображаются во всплывающем окне при вводе строк и полностью действуют даже без вызова функции mainloop:

```
C:\...\PP4E\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 виджет, моей первой реакцией было: «Зачем вообще здесь нужны переменные tkinter, если можно зарегистрировать обработчики щелчков на виджетах?» На первый взгляд связанные переменные могут показаться излишними, но они упрощают некоторые действия с графическим интерфейсом. Не буду просить принять это на веру, а постараюсь объяснить, почему.

Имейте в виду, что обработчик для флажка, указанный в параметре command, будет выполняться при каждом щелчке – при переключении и в выбранное, и в невыбранное состояние. Поэтому если нужно совершить действие немедленно после щелчка на флажке, как правило, в обработчике события требуется узнать текущее значение флажка. Поскольку у флажка нет метода «get», с помощью которого можно было бы получить текущее его значение, обычно требуется запрашивать ассоциированную переменную, чтобы узнать, включен флажок или выключен.

Кроме того, в некоторых графических интерфейсах пользователям разрешается устанавливать флажки без вызова обработчиков, зарегистрированных с помощью параметра command, и получать значения где-либо позже в программе. В таком сценарии переменные служат для автоматического запоминания состояний флажков. Представителем этого последнего подхода является метод report в сценарии demoCheck.

Конечно, можно и вручную запоминать состояние каждого флажка в обработчиках событий. В примере 8.23 ведется свой список состоя-

ний флажков, который вручную обновляется в обработчиках событий, определяемых с помощью параметра command.

Пример 8.23. PP4E\Gui\Tour\demo-check-manual.py

```
# флажки, сложный способ (без переменных)

from tkinter import *
states = []                     # изменение объекта – не имени
def onPress(i):                 # сохраняет состояния
    states[i] = not states[i]  # изменяет False->True, True->False

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(False)
root.mainloop()
print(states)                   # при выходе вывести все состояния
```

Здесь lambda-выражение передает индекс нажатой кнопки в списке states. Иначе для каждой кнопки потребовалось бы создавать отдельный обработчик. Здесь мы снова вынуждены использовать аргумент со значением по умолчанию, чтобы передать переменную цикла lambda-выражению. В противном случае все 10 сгенерированных функций получили бы значение переменной цикла, присвоенное ей в последней итерации цикла (щелчок на любом флажке изменял бы состояние десятого элемента в списке – причины такого поведения описываются в главе 7). При запуске этот сценарий создает окно с 10 флажками, как показано на рис. 8.27.

***Рис. 8.27.** Окно флажков с изменением состояний, производимым вручную*

Состояния флажков, поддерживаемые вручную, обновляются при каждом щелчке на флажках и выводятся при выходе из программы (формально, при возврате из вызова mainloop) – это список логических значений, которые можно было бы представить целыми числами 1 и 0, если бы потребовалось в точности имитировать оригинал:

```
C:\...\PP4E\Gui\Tour> python demo-check-manual.py
[False, False, True, False, True, False, False, False, True, False]
```

Такой способ действует, и его не столь уж трудно реализовать. Но связанные переменные tkinter заметно упрощают эту задачу, особенно если до какого-то момента в будущем нет необходимости проверять состояния флажков. Это проиллюстрировано в примере 8.24.

Пример 8.24. PP4E\Gui\Tour\demo-check-auto.py

```
# проверка состояния флажков, простой способ

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 следит библиотека tkinter
print([var.get() for var in states]) # вывести все состояния при выходе
                                     # (можно также реализовать с помощью
                                     # функции map и lambda-выражение)
```

Этот сценарий выводит такое же окно и действует точно так же, но здесь мы не передаем обработчики в параметре `command`, потому что библиотека tkinter автоматически отслеживает изменение состояний:

```
C:\...\PP4E\Gui\Tour> python demo-check-auto.py
[0, 0, 1, 1, 0, 0, 1, 0, 0, 1]
```

Смысл здесь в том, что необязательно связывать переменные с флажками, но если сделать это, то работать с графическим интерфейсом будет проще. Между прочим, генератор списков в самом конце этого сценария является эквивалентом следующим вызовам функции `map` со связанным методом или lambda-выражением в качестве аргумента:

```
print(list(map(IntVar.get, states)))
print(list(map(lambda var: var.get(), states)))
```

Хотя генераторы списков получили большое распространение в настоящее время, тем не менее то, какая форма наиболее понятна вам, может заметно зависеть от вашего... размера обуви.

Переключатели

Переключатели (или радиокнопки) обычно используются группами: так же, как для механических кнопок выбора станций в старых радиоприемниках, щелчок на одном виджете `Radiobutton` из группы автоматически делает невыбранными все кнопки, кроме той, на которой был выполнен последний щелчок. Иными словами, одновременно может быть выбрано не более одного виджета. В tkinter связывание всех переключателей из группы с уникальными значениями с одной и той же переменной гарантирует, что в каждый данный момент времени может быть выбрано не более одного переключателя.

Подобно флажкам и обычным кнопкам переключатели поддерживают параметр `command` для регистрации функции обратного вызова, обрабатывающей щелчок. 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 библиотеке tkinter, какое значение должна иметь ассоциированная переменная, когда выбирается тот или иной переключатель в группе. Поскольку несколько переключателей ассоциируется с одной и той же переменной, каждому переключателю должно соответствовать свое значение (это не просто схема с переключением между 1 и 0). Основы использования переключателей демонстрируются в примере 8.25.

Пример 8.25. PP4E\Gui\Tour\demoRadio.py

```
"создает группу переключателей, которые вызывают демонстрационные диалоги"

from tkinter import *          # импортировать базовый набор виджетов
from dialogTable import demos  # обработчики событий
from quitter import Quitter    # прикрепить к "себе" объект Quitter

class Demo(Frame):
    def __init__(self, parent=None, **options):
        Frame.__init__(self, parent, **options)
        self.pack()
        Label(self, text="Radio demos").pack(side=TOP)
        self.var = StringVar()
        for key in demos:
            Radiobutton(self, text=key,
                              command=self.onPress,
                              variable=self.var,
                              value=key).pack(anchor=NW)
        self.var.set(key)      # при запуске выбрать последний переключатель
        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. 8.28 изображено окно, которое создается при запуске этого сценария. Щелчок на любом из переключателей в этом окне вызывает обработчик command, запускает один из стандартных диалогов, с которыми мы познакомились выше, и автоматически делает невыбранным переключатель, на котором выполнялся щелчок перед этим. Как и флажки, переключатели здесь также компонуются; в данном сценарии они

прикрепляются к верхнему краю, располагаясь по вертикали, а затем выравниваются, прикрепляясь якорями к северо-западному углу отведенного им пространства.

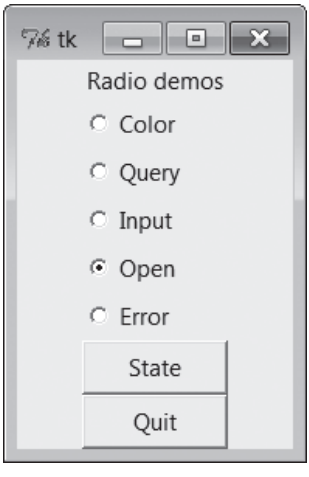


Рис. 8.28. *Сценарий demoRadio в действии*

Как и в примере с флажками, кнопка State служит для запуска метода `report` класса и вывода информации о текущем состоянии переключателей (выбранного переключателя). В отличие от примера с флажками, в этом сценарии выводятся также значения, возвращаемые диалогами, которые запускаются щелчками на переключателях. Ниже показано, как выглядит поток `stdout` после нескольких щелчков на переключателях – информация о состоянии выделена полужирным шрифтом:

```
C:\...\PP4E\Gui\Tour> python demoRadio.py
you pressed Input
result: 3.14
Input
you pressed Open
result: C:/PP4thEd/Examples/PP4E/Gui/Tour/demoRadio.py
Open
you pressed Query
result: yes
Query
```

Переключатели и переменные

Так зачем здесь нужны переменные? Первое, у переключателей нет метода «get», который позволил бы получить значение выбора. Еще более важно то, что в группах переключателей именно параметры `value` и `variable` обслуживают режим выбора единственного варианта. Вообще, работа переключателей обеспечивается тем, что вся группа ассо-

циируется с одной и той же переменной tkinter и при этом все переключатели имеют различные значения. Чтобы до конца разобраться в этом, нужны еще некоторые сведения о том, как взаимодействуют переключатели и переменные.

Как мы уже видели, при изменении состояния виджета изменяется ассоциируемая с ним переменная tkinter, и наоборот. Но также верно и то, что любое изменение переменной автоматически изменяет каждый виджет, с которым она связана. При работе с переключателями щелчок на одном из них устанавливает значение совместно используемой переменной, которая, в свою очередь, оказывает воздействие на другие переключатели, ассоциированные с этой переменной. При условии, что все переключатели имеют различные значения, это вызывает ожидаемый эффект. Когда в результате щелчка на переключателе значение совместно используемой переменной изменяется на значение выбранного переключателя, все остальные переключатели оказываются невыбранными, потому что значение переменной не совпадает с их значениями.

Это правило действует в обоих направлениях: когда пользователь выбирает переключатель – он неявно изменяет значение совместно используемой переменной; когда сценарий изменяет значение переменной, – он изменяет состояние переключателей. Например, когда сценарий в примере 8.25 на этапе инициализации присваивает совместно используемой переменной последнее значение последнего переключателя (вызовом `self.var.set`), он выбирает последний переключатель, а остальные автоматически становятся невыбранными. В результате изначально будет выбран только один переключатель. Если бы в переменную была записана строка, не являющаяся именем какого-либо демонстрационного диалога (например, ‘ ‘), все переключатели при запуске оказались бы невыбранными.

Это довольно тонкий волновой эффект, но его будет проще понять, представив картину с другой стороны: если в группе переключателей, связанных с одной и той же переменной, назначить нескольким переключателям одно и то же значение, то при щелчке на любом из них все они будут автоматически выбраны. Рассмотрим пример 8.26 и рис. 8.29. При запуске сценария не выбран ни один переключатель (так как совместно используемая переменная инициализирована значением, не соответствующим ни одному из значений переключателей), но поскольку переключатели 0, 3, 6 и 9 – имеют значение 0 (остаток от деления на 3), при выборе любого из них выбираются они все.

Рис. 8.29. *Переключатели испортились?*

Пример 8.26. PP4E\Gui\Tour\demo-radio-multi.py

```
# посмотрите, что произойдет, если несколько переключателей
# будут иметь одно и то же 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)
var.set(' ')           # все переключатели сделать невыбранными
root.mainloop()
```

Если теперь щелкнуть на любом из переключателей 1, 4 или 7, будут выбраны все три, а предыдущие выбранные окажутся сброшенными (их значение не равно «1»). Обычно это не то, что требуется, – переключатели как правило используются для представления групп с возможностью выбора единственного варианта (возможность выбора сразу нескольких вариантов реализуется с помощью флажков). Если вы хотите, чтобы переключатели действовали, как им положено, следите за тем, чтобы всем переключателям была назначена одна и та же переменная, но разные значения. Например, в сценарии `demoRadio` имя демонстрационного диалога дает естественное уникальное значение для каждой кнопки.

Переключатели без переменных

Строго говоря, в этом примере мы могли бы обойтись и без переменных tkinter. В примере 8.27 также реализована модель с одним выбором, но без переменных, – путем выбора и сброса элементов в группе вручную, в обработчике события. При каждом событии щелчка на переключателе вызывается метод `deselect` для всех объектов в группе и метод `select` для того переключателя, на котором был выполнен щелчок.

Пример 8.27. PP4E\Gui\Tour\demo-radio-manual.py

```
"""
переключатели, сложный способ (без переменных)
обратите внимание, что метод deselect переключателя просто устанавливает пустую
строку в качестве его значения, поэтому нам по-прежнему требуется присвоить
переключателям уникальные значения или 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)

onPress(0)              # первоначально выбрать первый переключатель
root.mainloop()
print(state)            # вывести информацию о состоянии перед выходом
```

Этот сценарий создает такое же окно с 10 переключателями, как на рис. 8.29, но реализует интерфейс с единственным выбором, причем текущее состояние хранится в глобальной переменной Python, значение которой выводится при завершении сценария. Все это библиотека tkinter может сделать вместо вас, если использовать связанную переменную tkinter и уникальные значения, как показано в примере 8.28.

Пример 8.28. PP4E\Gui\Tour\demo-radio-auto.py

```
# переключатели, простой способ

from tkinter import *
root = Tk()             # IntVar также можно использовать
var = IntVar(0)         # выбрать 0-й переключатель при запуске
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 информацию о состоянии перед выходом
```

Этот сценарий действует точно так же, но вводить и отлаживать его значительно проще. Обратите внимание, что в этом сценарии переключатели связываются с переменной типа `IntVar`, целочисленным собратом `StringVar`, которая инициализируется нулевым значением (которое также является значением по умолчанию) – если значения переключателей 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, пока переключатели отображаются на экране. Присвойте ссылку на объект глобальной переменной модуля, запомните в структуре данных с длительным временем существования или сохраните как атрибут долгоживущего объекта класса, как сделано в сценарии `demoRadio`. Просто сохраните ссылку на него где-нибудь. В этом случае у вас всегда будет возможность тем или иным способом получить

информацию о состоянии, и вас вряд ли когда-нибудь коснется то, о чем я хочу рассказать.

В текущей версии tkinter классы переменных обладают деструктором `__del__`, который автоматически сбрасывает созданную переменную `Tk`, когда уничтожается объект Python (то есть утилизируется сборщиком мусора). В итоге все ваши переключатели могут оказаться невыбранными, если объект переменной будет утилизирован, по крайней мере, до того момента, когда очередной щелчок мышью установит новое значение переменной `Tk`. В примере 8.29 демонстрируется ситуация, в которой это может случиться.

Пример 8.29. PP4E\Gui\Tour\demo-radio-clear.py

```
# берегите переменные переключателей (о чем действительно легко можно забыть)

from tkinter import *
root = Tk()

def radio1():           # локальные переменные являются временными
    #global tmp        # сделав их глобальными, вы решите проблему
    tmp = IntVar()
    for i in range(10):
        rad = Radiobutton(root, text=str(i), value=i, variable=tmp)
        rad.pack(side=LEFT)
    tmp.set(5)          # выбрать 6-й переключатель

radio1()
root.mainloop()
```

Кажется, что первоначально должен быть выбран переключатель «5», но этого не происходит. Локальная переменная `tmp` уничтожается при выходе из функции, переменная `Tk` сбрасывается и значение 5 теряется (все переключатели оказываются невыбранными). Тем не менее эти переключатели прекрасно работают, если попробовать выполнять на них щелчки мышью, поскольку при этом переменная `Tk` переустанавливается. Если раскомментировать инструкцию `global`, кнопка 5 будет появляться в выбранном состоянии, как и задумывалось.

Однако в версии Python 3.X это явление, похоже, приобрело дополнительные отрицательные черты: в этой версии переключатель «5» не только не выбирается изначально, но и перемещение указателя мыши над невыбранными переключателями порождает эффект незаказанного выбора многих их них, пока не будет выполнен щелчок мышью. (В версии 3.X также требуется инициализировать строковую переменную `StringVar`, совместно используемую переключателями, как мы делали это в предыдущих примерах; в противном случае переменная получит пустую строку, как значение по умолчанию, что переведет все переключатели в выбранное состояние!)

Конечно, это нетипичный пример – в таком виде невозможно узнать, какая кнопка нажата, потому что переменная не сохраняется (и параметр `command` не установлен). Довольно бессмысленно использовать группу переключателей, если позднее нельзя получить значение выбора. Фактически это настолько невразумительно, что я отсылаю вас к примеру *demo-radio-clear2.py* в пакете примеров для книги, 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 предупреждал.

Ползунки

Ползунки («scales» или «sliders») используются для выбора значения из диапазона чисел. Перемещение ползунка с помощью перетаскивания или щелчка мышью изменяет значение виджета в диапазоне целых чисел и запускает обработчик, если он зарегистрирован.

Подобно флажкам и переключателям ползунки обладают параметром `command` для регистрации управляемого событиями обработчика, выполняемого немедленно при перемещении ползунка, а также параметром `variable` для связи с переменной tkinter, которая позволяет в любой момент времени получить или установить положение ползунка. Обрабатывать значение можно сразу же после его установки или позже.

Кроме того, у ползунков есть третий способ обработки – методы `get` и `set`, с помощью которых можно непосредственно обращаться к значению виджета, не связывая с ним переменную. Поскольку обработчики, регистрируемые с помощью параметра `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 8.30 приводится сценарий, который создает два ползунка – горизонтальный и вертикальный, связанные между собой через ассоциированную переменную, что позволяет их синхронизировать.

Пример 8.30. PP4E\Gui\Tour\demoScale.py

```
"создает два связанных ползунка для запуска демонстрационных диалогов"

from tkinter import *          # импортировать базовый набор виджетов
from dialogTable import demos # обработчики событий
from quitter import Quitter    # прикрепить к "себе" объект Quitter

class Demo(Frame):
    def __init__(self, parent=None, **options):
        Frame.__init__(self, parent, **options)
        self.pack()
        Label(self, text="Scale demos").pack()
        self.var = IntVar()
        Scale(self, label='Pick demo number',
```

```
                    command=self.onMove,   # перехватывать перемещения
                    variable=self.var,     # отражает положение
                    from_=0, to=len(demos)-1).pack()
        Scale(self, label='Pick demo number',
                    command=self.onMove,   # перехватывать 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)
        demo = list(demos.values())[pos] # отображение позиции на ключ
                                         # (представление в версии 3.X)
        print(demo())                    # или
                                         # demos[ list(demos.keys())[pos] ]()
    def report(self):
        print(self.var.get())

if __name__ == '__main__':
    print(list(demos.keys()))
    Demo().mainloop()
```

Кроме доступа к значениям и регистрации обработчиков у ползунков имеются параметры, соответствующие понятию диапазона выбираемых значений, большинство из которых продемонстрировано в этом примере:

- Параметр `label` позволяет определить текст, появляющийся рядом со шкалой, параметр `length` позволяет определить начальный размер в пикселях, а параметр `orient` – направление.
- Параметры `from_` и `to` позволяют определить минимальное и максимальное значения шкалы (обратите внимание, что `from` в языке Python является зарезервированным словом, а `from_` – нет).
- Параметр `tickinterval` позволяет определить количество единиц измерения между отметками, наносимыми рядом со шкалой через равные интервалы (значение 0 по умолчанию означает, что отметки не выводятся).
- Параметр `resolution` позволяет определить количество единиц, на которое изменяется значение ползунка при каждом перетаскивании или щелчке левой кнопки мыши (по умолчанию 1).

- Параметр `showvalue` позволяет определить, должно ли отображаться текущее значение рядом с ползунком (по умолчанию `showvalue=YES`, то есть отображается).

Обратите внимание, что ползунки тоже прикрепляются к своим контейнерам, как и прочие виджеты tkinter. Посмотрим, как эти параметры используются на практике. На рис. 8.30 изображено окно, создаваемое этим сценарием в Windows 7 (в Unix и Mac получается аналогичная картина).

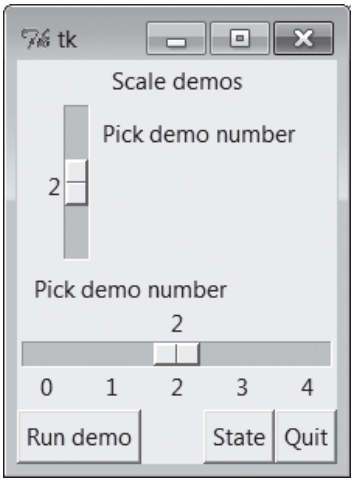


***Рис. 8.30.** Сценарий demoScale в действии*

Для наглядности кнопка State выводит текущие значения ползунков, а Run demo – запускает те же стандартные диалоги, используя целочисленные значения ползунков в качестве индекса таблицы `demos`. Сценарий также регистрирует обработчик `command`, который вызывается при каждом перемещении ползунка по любой из шкал и выводит новые значения ползунков. Ниже приводятся сообщения, отправленные в поток `stdout` после нескольких перемещений, с информацией о запускаемых демонстрационных диалогах (курсив) и о значениях ползунка (полужирный):

```
C:\...\PP4E\Gui\Tour> python demoScale.py
['Color', 'Query', 'Input', 'Open', 'Error']
in onMove 0
in onMove 0
in onMove 1
1
in onMove 2
You picked 2
123.0
```

```
in onMove 3
3
You picked 3
C:/Users/mark/Stuff/Books/4E/PP4E/dev/Examples/PP4E/Launcher.py
```

Ползунки и переменные

Как можно догадаться, ползунки предлагают различные способы обработки своих значений: непосредственно в обработчике события перемещения или позже, путем получения текущего положения через переменные или вызовы методов. В действительности переменные tkinter вообще не нужны для программирования ползунков – достаточно зарегистрировать обработчик события перемещения или вызывать метод `get` ползунка, чтобы при необходимости получать значение шкалы, как показано в более простом примере 8.31.

Пример 8.31. PP4E\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. 8.31 изображены два экземпляра этой программы, запущенные в Windows, окно одной из них растянуто, а другой – нет (ползунки

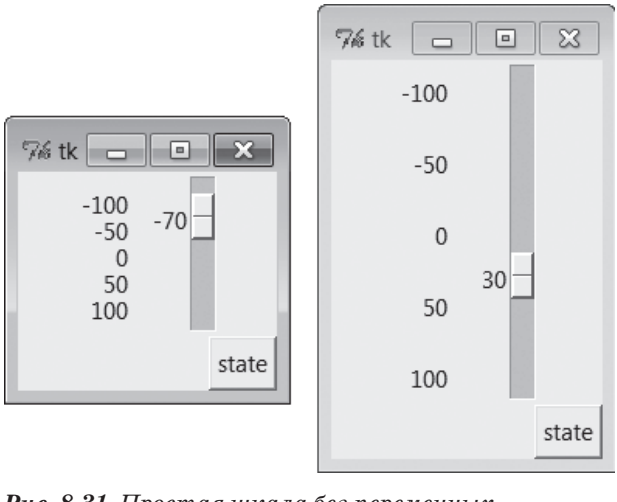


***Рис. 8.31.** Простая шкала без переменных*

настроены так, чтобы они растягивались по вертикали). Шкала имеет диапазон значений от –100 до 100, использует параметр `resolution` для изменения текущей позиции на 10 единиц вверх или вниз при каждом перемещении, а параметр `tickinterval` установлен так, чтобы рядом со шкалой отображались метки с шагом 50. Если щелкнуть на кнопке State в окне этого сценария, будет вызван метод `get` шкалы для получения и вывода текущего значения без всяких переменных или обратных вызовов:

```
C:\...\PP4E\Gui\Tour> python demo-scale-simple.py
0
60
-70
```

Честно говоря, единственный повод использовать переменные 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 состояния виджета влечет за собой изменение его переменной, а изменение переменной изменяет состояние всех связанных с ней виджетов. Другими словами, это означает, что перемещение ползунка обновляет переменную, которая, в свою очередь, может обновить все другие виджеты, 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 подобный прием синхронизации ползунков (граничащий с глубокой магией), но если задуматься, то это очень мощное средство. Связывая несколько виджетов с помощью переменных tkinter, можно автоматически синхронизировать их и избежать необходимости вручную корректировать значения виджетов в обработчиках событий. С другой стороны, синхронизацию можно обеспечить и без совместно используемой переменной, вызывая метод `set` одного ползунка из обработчика другого. Такое изменение вручную я оставляю читателю в качестве самостоятельного упражнения. Что для одного глубокая магия, для другого может оказаться привычной рутиной.

Три способа использования графических интерфейсов

После того как мы создали ряд похожих программ для запуска демонстрационных примеров, напишем несколько объединяющих их сценариев верхнего уровня. Поскольку демонстрационные примеры были

написаны так, что их можно использовать в виде классов или сценариев, они могут быть использованы как компоненты, прикрепляемые к фреймам, могут запускаться в собственных окнах верхнего уровня или как самостоятельные программы. Все три варианта иллюстрируют принцип многократного использования программного кода в действии.

Прикрепление к фреймам

Чтобы проиллюстрировать иерархическое построение графического интерфейса в более крупном масштабе, чем это делалось до сих пор, пример 8.32 объединяет все четыре сценария панелей запуска диалогов из этой главы в одном контейнере. В нем повторно используется программный код примеров 8.9, 8.22, 8.25 и 8.30.

Пример 8.32. PP4E\Gui\Tour\demoAll-frm.py

```
"""
4 класса демонстрационных компонентов (вложенных фреймов) в одном окне;
в одном окне присутствуют также 5 кнопок Quitter, причем щелчок на любой из
них приводит к завершению программы; графические интерфейсы могут повторно
использоваться, как фреймы в контейнере, независимые окна или процессы;
"""

from tkinter import *
from quitter import Quitter
demoModules = ['demoDlg', 'demoCheck', 'demoRadio', 'demoScale']
parts = []

def addComponents(root):
    for demo in demoModules:
        module = __import__(demo) # импортировать по имени в виде строки
        part = module.Demo(root)                    # прикрепить экземпляр
        part.config(bd=2, relief=GROOVE)            # или передать параметры
                                                    # конструктору Demo()
        part.pack(side=LEFT, expand=YES, fill=BOTH) # растягивать
                                                    # вместе с окном
        parts.append(part)                          # добавить в список

def dumpState():
    for part in parts:
        print(part.__module__ + ':', end=' ')
        if hasattr(part, 'report'):                 # вызвать метод report,
            part.report()                           # если имеется
        else:
            print('none')

root = Tk()                                         # явно создать корневое окно
root.title('Frames')
Label(root, text='Multiple Frame demo', bg='white').pack()
Button(root, text='States', command=dumpState).pack(fill=X)
```

```
Quitter(root).pack(fill=X)
addComponents(root)
root.mainloop()
```

Поскольку все четыре демонстрационные панели запуска реализованы в виде фреймов, которые могут прикрепляться к родительским виджетам, объединить их в одном графическом интерфейсе намного проще, чем вы думаете. Для этого нужно лишь передать один и тот же родительский виджет (в данном случае окно `root`) во все четыре вызова конструкторов демонстрационных примеров, после чего скомпоновать и настроить созданные демонстрационные объекты желаемым образом. На рис. 8.32 показано, как выглядит результат – одно окно, в которое встроены экземпляры всех четырех знакомых нам демонстрационных панелей запуска диалогов. В данном примере все четыре встроенных панели изменяют свои размеры при изменении размеров окна (попробуйте убрать параметр `expand=YES`, чтобы панели сохраняли свои размеры постоянными).

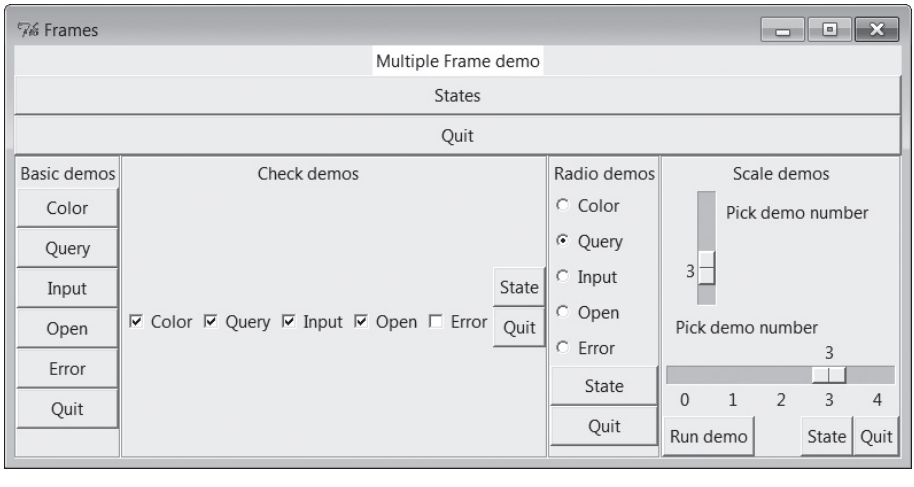


Рис. 8.32. *demoAll_frm: вложенные фреймы*

Конечно, этот пример является искусственным, но он демонстрирует мощь приема композиции при создании больших графических интерфейсов. Если на месте демонстрационных объектов вы представите себе что-нибудь более полезное, например текстовый редактор, калькулятор или часы, то лучше поймете значение этого примера.

Кроме фреймов демонстрационных объектов это составное окно содержит не менее пяти экземпляров написанной ранее кнопки `Quitter` (любая из них может завершить работу этой программы) и кнопку States для вывода текущих значений сразу всех встроенных демонстрационных объектов (она вызывает метод `report` каждого объекта, у которого

он есть). Ниже приводится пример вывода в потоке stdout после взаимодействия с виджетами в этом окне; вывод, полученный в результате щелчка на кнопке States, выделен полужирным шрифтом:

```
C:\...\PP4E\Gui\Tour> python demoAll_frm.py
in onMove 0
in onMove 0
demoDlg: none
demoCheck: 0 0 0 0 0
demoRadio: Error
demoScale: 0
you pressed Input
result: 1.234
in onMove 1
demoDlg: none
demoCheck: 1 0 1 1 0
demoRadio: Input
demoScale: 1
you pressed Query
result: yes
in onMove 2
You picked 2
None
in onMove 3
You picked 3
C:/Users/mark/Stuff/Books/4E/PP4E/dev/Examples/PP4E/Launcher.py
3
Query
1 1 1 1 0
demoDlg: none
demoCheck: 1 1 1 1 0
demoRadio: Query
demoScale: 3
```

Импортирование по имени в виде строки

Единственным хитроумным приемом в этом сценарии является использование встроенной функции __import__, импортирующей модуль по его имени в виде строки. Взглянем на следующие две строки из функции addComponents сценария:

```
module = __import__(demo) # импортировать по имени в виде строки
part = module.Demo(root)  # прикрепить экземпляр, созданный его классом Demo
```

Они эквивалентны следующим строкам:

```
import 'demoDlg'
part = 'demoDlg'.Demo(root)
```

Однако последние две инструкции являются недопустимыми в языке Python – имя модуля в инструкциях импорта должно быть идентификатором Python, а не строковым значением. Кроме того, имена модулей

в инструкциях импорта интерпретируются буквально (то есть не вычисляются), и в точечной нотации идентификаторы должны выражать объект (а не строку с именем). Для обеспечения общности функция `addComponents` обходит список строк с именами и с помощью `__import__` импортирует и возвращает модуль, идентифицируемый каждой строкой. Фактически действие цикла `for` эквивалентно следующим инструкциям:

```
import demoDlg, demoRadio, demoCheck, demoScale
part = demoDlg.Demo(root)
part = demoRadio.Demo(root)
part = demoCheck.Demo(root)
part = demoScale.Demo(root)
```

Использование в сценарии списка строк с именами упрощает изменение набора встраиваемых демонстраций – достаточно лишь изменить список, не трогая выполняемый код. Кроме того, такая реализация, управляемая данными, оказывается более компактной, менее избыточной и проще в отладке и в сопровождении. Между прочим, имеется еще одна возможность импортировать модули по их именам в виде строк, динамически создавая и выполняя инструкции импорта, как показано ниже:

```
for demo in demoModules:
    exec('from %s import Demo' % demo) # сконструировать и выполнить from
    part = eval('Demo')(root)          # получить ссылку на импортированный
                                       # объект по его имени в виде строки
```

Функция `exec` компилирует и выполняет строку с инструкцией Python (в данном случае `from`, загружающей класс `Demo` из модуля). Она действует, как если бы вместо вызова функции `exec` в исходный текст была вставлена строка с инструкцией. Следующий фрагмент позволяет добиться того же эффекта, но конструируя инструкцию `import`:

```
for demo in demoModules:
    exec('import %s' % demo)     # сконструировать и выполнить import
    part = eval(demo).Demo(root) # получить ссылку на объект модуля
                                 # также по имени в виде строки
```

Так как функции `exec`/`eval` поддерживают любые инструкции Python, этот прием оказывается более универсальным, чем использование функции `__import__`, но он может замедлять работу, потому что при этом требуется производить синтаксический анализ строк программного кода перед их выполнением.[1] Такое замедление может быть несущественным для графических интерфейсов – пользователи работают значительно медленнее, чем синтаксические анализаторы.

---

[1] Как будет показано дальше, `exec` может представлять опасность, если выполняет строки, полученные от пользователей или по сети. Это не относится к жестко определенным строкам в данном примере.

Настройка на этапе конструирования

Еще одна альтернатива, о которой следует упомянуть: обратите внимание, как в примере 8.32 выполняется настройка и компоновка каждого прикрепляемого демонстрационного фрейма:

```
def addComponents(root):
    for demo in demoModules:
        module = __import__(demo) # импортировать по имени в виде строки
        part = module.Demo(root)                    # прикрепить экземпляр
        part.config(bd=2, relief=GROOVE)            # или передать параметры
                                                    # конструктору Demo()
        part.pack(side=LEFT, expand=YES, fill=BOTH) # растягивать
                                                    # вместе с окном
```

Однако благодаря тому что демонстрационные классы поддерживают параметры настройки, используя аргумент `**options`, мы могли бы выполнять настройки прямо на этапе создания. Например, если изменить реализацию сценария, как показано ниже, он воспроизведет несколько отличающееся окно, изображенное на рис. 8.33 (для иллюстрации несколько растянутое по горизонтали; вы найдете эту реализацию в файле *demoAll-frm-ridge.py* в пакете с примерами):

```
def addComponents(root):
    for demo in demoModules:
        module = __import__(demo) # импортировать по имени в виде строки
        part = module.Demo(root, bd=6, relief=RIDGE) # прикрепить, настроить
        part.pack(side=LEFT, expand=YES, fill=BOTH)  # экземпляр так, чтобы он
                                                     # растягивался с окном
```

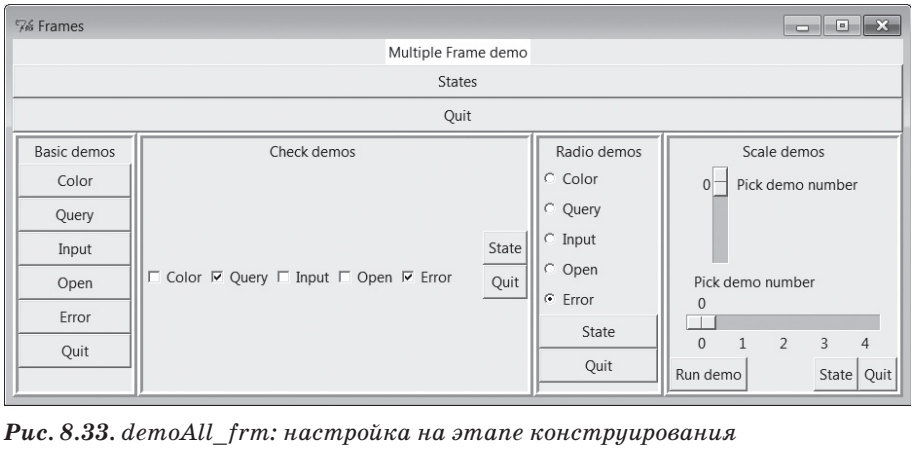


Рис. 8.33. *demoAll_frm: настройка на этапе конструирования*

Поскольку демонстрационные классы одновременно являются подклассами `Frame` и поддерживают привычный протокол передачи аргументов конструкторов, они являются настоящими виджетами – специализи-

рованными фреймами tkinter, реализующими прикрепляемые пакеты виджетов, и обеспечивающими гибкие возможности настройки.

В главе 7 было показано, что такое *прикрепление* вложенных фреймов является лишь одним из способов повторного использования программного кода реализации графических интерфейсов, оформленного в виде классов. Столь же просто можно организовывать такие интерфейсы путем создания *подклассов*, а не встраивания. Однако в данном случае нас больше интересует применение существующего пакета виджетов, чем его изменение, поэтому прием встраивания полностью соответствует нашим потребностям. В следующих двух разделах показаны еще два способа представления таких пакетов виджетов пользователям – в виде независимых окон и в виде автономных программ.

Независимые окна

Когда есть набор классов компонентов, реализованных в виде фреймов, годится любой родительский элемент – и фреймы, и новые окна верхнего уровня. В примере 8.33 все четыре объекта демонстрационных панелей прикрепляются к собственным независимым окнам `Toplevel`, а не к одному и тому же контейнеру.

Пример 8.33. PP4E\Gui\Tour\demoAll-win.py

```
"""
4 демонстрационных класса в независимых окнах верхнего уровня;
не процессы: при завершении одного щелчком на кнопке Quit завершаются все
остальные, потому что все окна выполняются в одном и том же процессе; здесь
первое окно Tk создается вручную, иначе будет создано пустое окно
"""

from tkinter import *
demoModules = ['demoDlg', 'demoRadio', 'demoCheck', 'demoScale']

def makePopups(modnames):
    demoObjects = []
    for modname in modnames:
        module = __import__(modname) # импортировать по имени в виде строки
        window = Toplevel()          # создать новое окно
        demo = module.Demo(window)   # родительским является новое окно
        window.title(module.__name__)
        demoObjects.append(demo)
    return demoObjects

def allstates(demoObjects):
    for obj in demoObjects:
        if hasattr(obj, 'report'):
            print(obj.__module__, end=' ')
            obj.report()
```

```
root = Tk()                              # явно создать корневое окно
root.title('Popups')
demos = makePopups(demoModules)
Label(root, text='Multiple Toplevel window demo', bg='white').pack()
Button(root, text='States', command=lambda: allstates(demos)).pack(fill=X)
root.mainloop()
```

Мы уже встречались с классом `Toplevel` – каждый его экземпляр создает на экране новое окно. Получаемый результат изображен на рис. 8.34 – каждая демонстрационная панель выполняется не в общем, а в собственном окне.

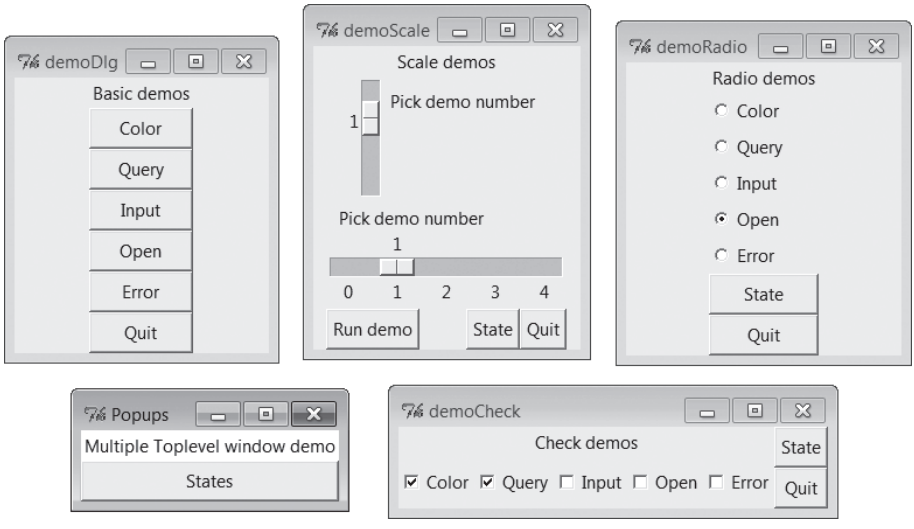


Рис. 8.34. *demoAll_win: новые окна Toplevel*

Главное корневое окно на этом рисунке находится в левом нижнем углу. На нем есть кнопка States, которая вызывает метод `report` каждого демонстрационного объекта, выводя в `stdout` примерно такой текст:

```
C:\...\PP4E\Gui\Tour> python demoAll_win.py
in onMove 0
in onMove 0
in onMove 1
you pressed Open
result: C:/Users/mark/Stuff/Books/4E/PP4E/dev/Examples/PP4E/Launcher.py
demoRadio Open
demoCheck 1 1 0 0 0
demoScale 1
```

Как было показано ранее в этой главе, окна `Toplevel` функционируют независимо друг от друга, но в действительности они не являются отдельными программами. Закрытие любого из окон на рис. 8.34 щелч-

ком на кнопке X в правом верхнем углу закрывает только это окно. Но попытка закрыть окно щелчком на кнопке Quit или на кнопке X в главном окне закроет их *все* и завершит все приложение, потому что все они выполняются в одном и том же программном процессе. В некоторых приложениях это приемлемо, но не во всех. Чтобы обеспечить настоящую независимость, необходимо порождать дочерние процессы, как показано в следующем разделе.

Запуск программ

Для обеспечения большей независимости в примере 8.34 каждая из демонстрационных панелей запускается, как независимая программа (процесс), с помощью модуля launchmodes, который мы написали в конце главы 5. Это работает потому, что все демонстрационные примеры были написаны и как импортируемые классы, и как выполняемые сценарии. При запуске их таким образом каждая из них получает имя __main__, потому что являются отдельными, независимыми программами; а это, в свою очередь, приводит к запуску mainloop в конце каждого файла.

Пример 8.34. PP4E\Gui\Tour\demoAll-prg.py

```
"""
4 демонстрационных класса, выполняемых как независимые процессы: команды;
если теперь одно окно будет завершено щелчком на кнопке Quit, остальные
продолжат работу; в данном случае не существует простого способа вызвать все
методы report (впрочем, для организации взаимодействий между процессами можно
было бы воспользоваться сокетами и каналами), а кроме того, некоторые способы
запуска могут сбрасывать поток stdout дочерних программ и разрывать связь между
родителем и потомком;
"""

from tkinter import *
from PP4E.launchmodes import PortableLauncher
demoModules = ['demoDlg', 'demoRadio', 'demoCheck', 'demoScale']

for demo in demoModules:                    # смотрите главу 5
    PortableLauncher(demo, demo + '.py')() # запуск в виде программ верхнего
                                           # уровня

root = Tk()
root.title('Processes')
Label(root, text='Multiple program demo: command lines', bg='white').pack()
root.mainloop()
```

Перед запуском этого сценария убедитесь, что корневой каталог с примерами *PP4E* находится в пути поиска модулей (например, включен в переменную окружения PYTHONPATH) – он импортирует модуль из другого подкаталога. Как видно из рис. 8.35, создаваемый данным сценарием интерфейс 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 продолжают работу. Демонстрационные программы будут выполняться, даже если закрыть окно родительского процесса. В Windows окно командной строки, откуда был запущен этот сценарий, снова становится активным и готовым для ввода следующей команды, притом, что дочерние программы продолжают выполняться. Здесь мы повторно использовали программный код демонстрационных примеров, запуская их как самостоятельные программы, а не как модули.

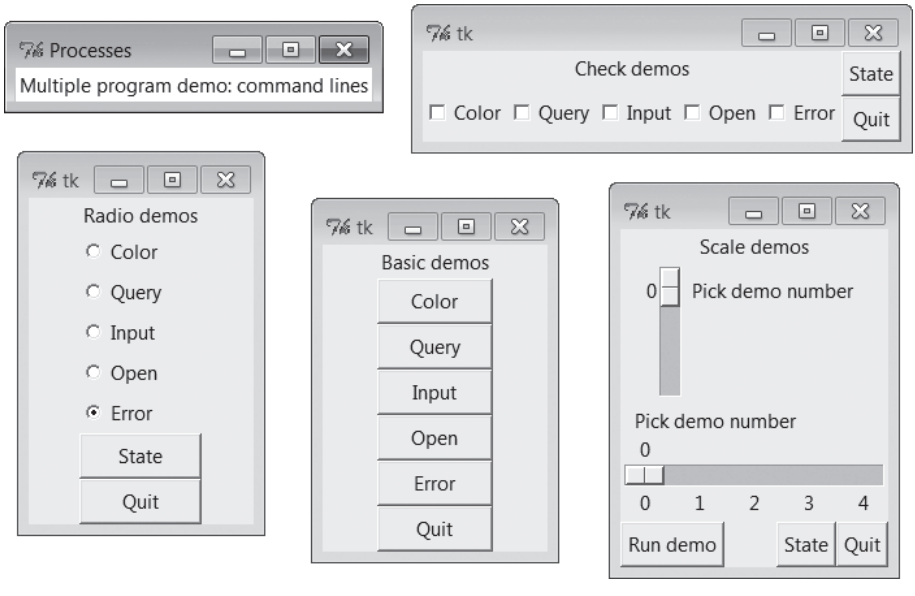


Рис. 8.35. *demoAll_prg: независимые программы*

Запуск графических интерфейсов как самостоятельных программ другими способами: модуль multiprocessing

Если вернуться к главе 5 и рассмотреть реализацию модуля запуска процессов переносимым способом, использованным в примере 8.34, можно заметить, что в Windows он использует функцию `os.spawnv`, а в других системах – `os.fork/exec`. То есть графические интерфейсы в данном примере запускаются выполнением *команд оболочки*. Эти способы прекрасно справляются со своей задачей, но, как мы узнали в главе 5, они входят в состав более обширного набора инструментов запуска программ, в число которых также входят `os.popen`, `os.system`, `os.startfile` и модули `subprocess` и `multiprocessing`. Эти инструменты могут отличаться деталями подключения к окну консоли, реакцией на завершение родительского процесса и так далее.

Например, модуль multiprocessing, с которым мы познакомились в главе 5, предоставляет похожий переносимый способ запуска других графических интерфейсов в виде независимых процессов, как показано в примере 8.35. Если запустить его, он воспроизведет точно такое же окно, как на рис. 8.35, но с другими метками в главном окне.

Пример 8.35. PP4E\Gui\Tour\demoAll-prg-multi.py

```
"""
4 демонстрационных класса, выполняемых как независимые процессы:
multiprocessing;
модуль multiprocessing позволяет запускать только именованные функции
с аргументами – он не может работать с lambda-выражениями, поскольку в Windows
они не могут быть сериализованы (глава 5); кроме того, модуль multiprocessing
имеет собственные инструменты взаимодействий между процессами, такие как каналы;
"""

from tkinter import *
from multiprocessing import Process
demoModules = ['demoDlg', 'demoRadio', 'demoCheck', 'demoScale']

def runDemo(modname):               # запускается в новом процессе
    module = __import__(modname)    # создать GUI с нуля
    module.Demo().mainloop()

if __name__ == '__main__':
    for modname in demoModules:     # только в __main__!
        Process(target=runDemo, args=(modname,)).start()

    root = Tk()                     # граф. интерфейс родительского процесса
    root.title('Processes')
    Label(root, text='Multiple program demo: multiprocessing',
                bg='white').pack()
    root.mainloop()
```

При запуске в Windows эта версия имеет только следующие функциональные отличия:

- Вывод дочерних процессов отображается в том же окне консоли, откуда был запущен этот сценарий, включая вывод, порождаемый самими демонстрационными диалогами и всеми кнопками State.
- Сценарий не завершается, если хотя бы один дочерний процесс продолжает выполнение: окно консоли в данном примере блокируется при попытке закрыть окно главного процесса, пока дочерние процессы продолжают работу, если только не установить флаг daemon дочерних процессов в значение True перед их запуском, как было показано в главе 5, – в этом случае все дочерние процессы автоматически будут завершены вместе с их родителем (но родитель по-прежнему может пережить своих потомков).

Обратите также внимание, как мы запускаем простую именованную функцию в новом процессе Process. Как мы узнали в главе 5, в Windows в аргументе target допускается передавать только сериализуемые выполняемые объекты (то есть те, которые можно импортировать), поэтому мы не можем использовать lambda-выражения для передачи дополнительных данных, как мы обычно делали это в обработчиках событий tkinter. Следующие два варианта реализации будут терпеть неудачу в Windows:

```
Process(target=(lambda: runDemo(modname))).start() # оба терпят неудачу!

Process(target=(lambda: __import__(modname).Demo().mainloop())).start()
```

Мы не будем здесь пытаться реализовать сценарий запуска программ с графическим интерфейсом всеми возможными способами, но вы можете поэкспериментировать с ними самостоятельно, используя главу 5, как источник информации по этой теме. Хотя это решение и не везде применимо, тем не менее в целом смысл использования инструментов, таких как класс PortableLauncher, – скрыть большинство таких подробностей, что позволило бы нам практически забыть о них.

Обмен данными между программами

Запуск графических интерфейсов в виде самостоятельных программ – вершина независимости кода, но это затрудняет связь между компонентами. Например, поскольку демонстрационные примеры выполняются здесь в виде отдельных программ, нет простого способа вызвать все их методы report из окна запускающего сценария, изображенного слева вверху на рис. 8.35. Кнопки States теперь нет, и в поток stdout попадают только сообщения от экземпляра PortableLauncher:

```
C:\...\PP4E\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 функция os.spawnv, используемая в модуле launchmodes для запуска программ, полностью отключает поток stdout дочерней программы от родителя. В любом случае нет прямого способа одновременно вызвать методы report во всех демонстрационных программах – это отдельные программы, выполняющиеся в отдельных адресных пространствах, а не импортированные модули.

Однако существует возможность организовать вызов методов report в порожденных программах с помощью некоторых механизмов IPC, с которыми мы познакомились в главе 5. Например:

- Демонстрационные программы могут быть оснащены механизмом приема *сигнала*, в ответ на который они будут вызывать свой метод `report`.
- Они могут ждать появления в именованных или неименованных *каналах* строк запросов, посылаемых запускающей программой – запускающая программа `demoAll` стала бы фактически клиентом, а демонстрационные сценарии – серверами, отвечающими на запросы клиента.
- Независимые программы могут общаться подобным образом с помощью *сокетов*, универсальным инструментом, представленным в главе 5, который мы будем подробно изучать в четвертой части книги. Главный сценарий мог бы отправлять запрос на получение отчета и принимать ответ в тот же самый сокет (и контактировать с демонстрационными сценариями, выполняющимися удаленно).
- При использовании модуля `multiprocessing` становятся доступны его собственные инструменты IPC, такие как каналы и очереди объектов, представленные в главе 5, которые также можно было бы задействовать для организации обмена данными: демонстрационные сценарии могли бы также прослушивать каналы этого типа.

Исходя из управляемой событиями природы, программы с графическим интерфейсом должны избегать перехода в *состояние ожидания* – они не должны блокироваться, ожидая появления запросов в механизмах IPC, иначе они не будут откликаться на действия пользователя (и даже не смогут перерисовывать себя). Поэтому может потребоваться дополнить их потоками выполнения, обработчиками, вызываемыми по таймеру, выполнять операции чтения в неблокирующем режиме или использовать комбинации этих инструментов для периодической проверки таких входящих сообщений в каналах, fifo или сокетах. Как мы увидим далее, метод `after` из библиотеки tkinter, описываемый ближе к концу следующей главы, является идеальным средством для этого: он позволяет регистрировать функции обратного вызова для периодической проверки наличия входящих запросов.

Мы исследуем некоторые из имеющихся возможностей ближе к концу главы 10, после того как рассмотрим темы, связанные с созданием многопоточных графических интерфейсов. Но поскольку это выходит далеко за рамки представленных простых демонстрационных программ, я оставляю реализацию таких межпрограммных взаимодействий тем из читателей, кто лучше подготовлен к мышлению в категориях параллельных процессов.

Программирование, обеспечивающее повторное использование

Постскриптум: я реализовал все демонстрационные панели запуска, разворачиваемые в четырех последних примерах так, чтобы продемон-

стрировать различные способы использования их виджетов. Они разрабатывались без учета возможности многократного их использования – на самом деле они не слишком полезны вне контекста знакомства с виджетами в данной книге.

Так было задумано – большинством графических элементов tkinter легко пользоваться после изучения их интерфейсов, а библиотека tkinter в значительной мере сама обеспечивает значительную гибкость настройки. Но если бы я задумал реализовать классы флажков и переключателей, которые можно повторно использовать как универсальные библиотечные компоненты, их нужно было бы структурировать иным образом:

Лишние виджеты

Они не должны выводить ничего, кроме переключателей и флажков. В существующем виде в каждый демонстрационный пример для иллюстрации встроены кнопки State и Quit, но в действительности у каждого окна верхнего уровня должна быть только одна кнопка Quit.

Управление компоновкой

Они должны допускать различное расположение кнопок и вообще никак не компоновать себя (ни методом `pack`, ни методом `grid`). В понастоящему универсальных реализациях для повторного использования часто лучше передоверить вызывающей программе самой управлять размещением 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 вызывать обработчики при нажатии или позволить приложению получать их состояние позже.

В примере 8.36 демонстрируется один из способов реализации панелей флажков и переключателей как библиотечных компонентов. Он основывается на связывании переменных tkinter и для обеспечения простоты интерфейса требует, чтобы вызывающая программа использовала наиболее общий режим – получение информации о состоянии вместо обработчиков событий – чтобы упростить интерфейс.

Пример 8.36. PP4E\Gui\Tour\buttonbars.py

```
"""
классы панелей флажков и переключателей для приложений, которые запрашивают
информацию о состоянии позднее;
передается список вариантов выбора, вызывается метод state(), работа
с переменными выполняется автоматически
"""
```

```
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 [var.get() for var in self.vars]

class Radiobar(Frame):
    def __init__(self, parent=None, picks=[], side=LEFT, anchor=W):
        Frame.__init__(self, parent)
        self.var = StringVar()
        self.var.set(picks[0])
        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)

    def allstates():
        print(gui.state(), lng.state(), tgl.state())

    from quitter import Quitter
    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 вариантов выбора, которые должны появиться на панелях флажков или переключателей. Программный код самотестирования модуля, находящийся в конце, демонстрирует особенности использования этих классов. Если запустить этот при-

мер как самостоятельный сценарий, на экране появится окно верхнего уровня, изображенное на рис. 8.36, с двумя встроенными панелями Checkbar, одной панелью Radiobar, кнопкой Quitter для завершения, а также кнопкой Peek для вывода информации о состоянии панелей.

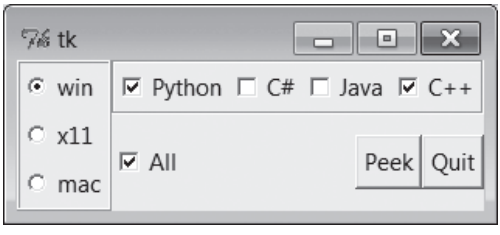


Рис. 8.36. *Окно самотестирования сценария buttonbars*

Ниже приводится содержимое стандартного потока вывода stdout после щелчка на кнопке Peek – результат вызова 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 замысловатые особенности реализации панелей переключателей и флажков. Например, при использовании таких классов высокого уровня можно полностью забыть о том, как нужно использовать связанные переменные, – достаточно просто создать объекты со списками вариантов выбора и затем вызывать их методы state. Если пойти этим путем до конца, можно прийти к библиотеке виджетов более высокого уровня, такой как пакет Pmw, упоминавшийся в главе 7.

С другой стороны, эти классы все же не готовы для универсального применения. Например, если потребуется выполнять действия при выборе вариантов, придется использовать другие интерфейсы верхнего уровня. К счастью, Python/tkinter предоставляют великое их множество. Далее в этой книге мы снова будем пользоваться комбинациями виджетов и снова будем применять приемы, представленные в этой главе, для создания более крупных графических интерфейсов. А сейчас последняя остановка в этой первой главе, посвященной экскурсии по виджетам, – фотолаборатория.

Изображения

В библиотеке tkinter графические изображения отображаются за счет создания независимых объектов PhotoImage или BitmapImage и прикрепления их к другим виджетов путем установки атрибута image. Кнопки, метки, холсты, текстовые виджеты и меню – все они могут выводить изображения, связывая таким способом готовые графические

объекты. Для иллюстрации сценарий в примере 8.36 выводит картинку на кнопке.

Пример 8.37. PP4E\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()
```

Трудно было бы придумать более простой пример: этот сценарий всего лишь создает объект `PhotoImage` для GIF-файла, хранящегося в другом каталоге, и связывает его с параметром `image` виджета `Button`. Результат изображен на рис. 8.37.

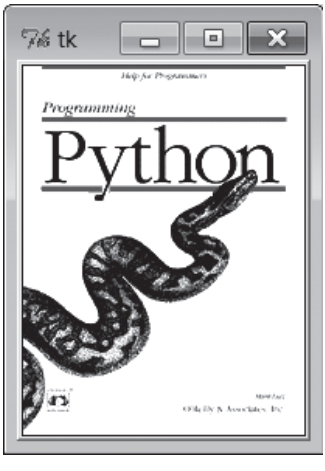


Рис. 8.37. *Сценарий imgButton в действии*

Объект `PhotoImage` и его собрат `BitmapImage` просто загружают графические файлы и позволяют прикреплять полученные изображения к другим типам виджетов. Чтобы открыть файл с картинкой, его имя необходимо передать в атрибуте `file` этих виджетов изображений. Несмотря на всю простоту, прикрепление изображений к кнопкам может найти применение во множестве ситуаций – в главе 9, например, мы будем использовать эту простую идею при реализации кнопок для панелей инструментов в нижней части окна.

Виджеты `Canvas` – универсальные поверхности для вывода графики, подробнее обсуждаемые в следующей главе, тоже могут выводить картинки. Забегая вперед, в качестве предварительного знакомства отмечу, что холсты (объекты `Canvas`) достаточно просты в обращении, чтобы привести их в примере. Пример 8.38 выводит окно, изображенное на рис. 8.38.

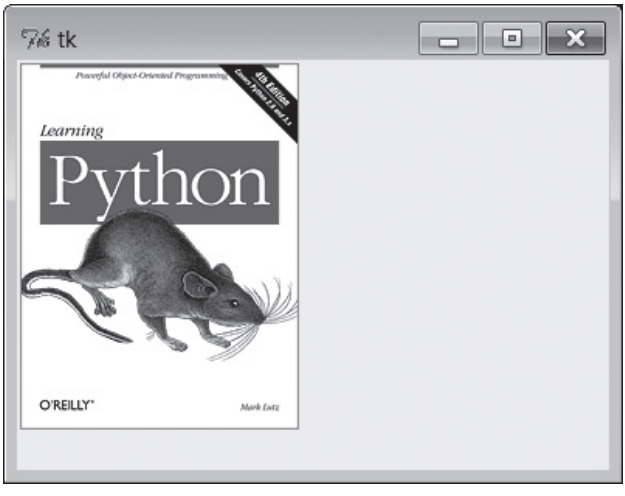


***Рис. 8.38.** Изображение на холсте*

Пример 8.38. PP4E\Gui\Tour\imgCanvas.py

```
gifdir = "../gifs/"
from tkinter import *
win = Tk()
img = PhotoImage(file=gifdir + "ora-lp4e.gif")
can = Canvas(win)
can.pack(fill=BOTH)
can.create_image(2, 2, image=img, anchor=NW)   # координаты x, y
win.mainloop()
```

Размеры кнопок автоматически изменяются в соответствии с размерами изображений, холсты свои размеры не изменяют (потому что в холсты можно добавлять объекты, как будет показано в главе 9). Чтобы размер холста соответствовал размерам изображения, нужно установить его размер, исходя из значений, возвращаемых методами `width` и `height` объектов изображений, как в примере 8.39. Эта версия сценария при необходимости делает холст больше или меньше, чем размер, устанавливаемый по умолчанию; позволяет передавать имя графического файла в аргументе командной строки и может использоваться в качестве простой утилиты просмотра изображений. Окно, создаваемое этим сценарием, изображено на рис. 8.39.

Пример 8.39. PP4E\Gui\Tour\imgCanvas2.py

```
gifdir = "../gifs/"
from sys import argv
from tkinter import *
filename = argv[1] if len(argv) > 1 else 'ora-lp4e.gif' # имя файла
                                                       # в командной строке?
```

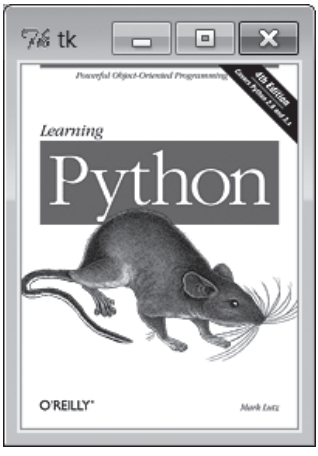


Рис. 8.39. *Изменение размера холста соответственно картинке*

```
win = Tk()
img = PhotoImage(file=gifdir + filename)
can = Canvas(win)
can.pack(fill=BOTH)
can.config(width=img.width(), height=img.height()) # размер соответственно
can.create_image(2, 2, image=img, anchor=NW)       # картинке
win.mainloop()
```

Чтобы просмотреть другое изображение, нужно запустить этот сценарий, передав ему имя другого файла (попробуйте сами, на своем компьютере):

```
C:\...\PP4E\Gui\Tour> imgCanvas2.py ora-ppr-german.gif
```

Вот и все. В главе 9 будет показано, как помещать изображения в элементы меню, в кнопки на панели инструментов, приведены другие примеры с объектом `Canvas` и дружественный к изображениям виджет `Text`. В последующих главах они встретятся в программе просмотра слайдов (PyView), графическом редакторе (PyDraw) и в других. В графических интерфейсах Python/tkinter очень легко добавлять графику.

Однако начав всерьез использовать графические изображения, вы наверняка наткнетесь на два подводных камня, о которых я хочу предупредить заранее:

Поддерживаемые типы файлов

В настоящее время виджет `PhotoImage` поддерживает только файлы форматов GIF, PPM и PGM, а `BitmapImage` поддерживает файлы растровых изображений *.xbm* в стиле X Window. В последующих версиях количество поддерживаемых форматов может расшириться, и,

конечно же, вы можете предварительно преобразовать свои изображения в указанные форматы. Но как будет показано далее в этой главе, поддержку дополнительных форматов изображений легко можно обеспечить с помощью открытого пакета PIL и его класса `PhotoImage`.

Берегите свои фотографии!

В отличие от других виджетов, tkinter изображение будет безвозвратно утрачено, если соответствующий объект изображения будет утилизирован сборщиком мусора. Это значит, что необходимо сохранять явные ссылки на объекты с графикой в течение всего времени, когда они могут понадобиться программе (например, присвоить их долгоживущей переменной, атрибуту объекта или компоненту структуры данных). Интерпретатор не сохраняет автоматически ссылку на графическое изображение, даже если оно связано с другими компонентами графического интерфейса, отображающими его. Кроме того, методы деструкторов объектов 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 еще более вероятен. В будущих версиях Python такое поведение может измениться; при этом существуют веские причины не держать в памяти большие графические файлы неопределенно долгое время. На данный же момент элементы с графикой таковы, что если не используются, то могут быть утеряны.

Развлечения с кнопками и картинками

Я хотел представить для этого раздела демонстрационный пример работы с графикой, который был бы одновременно забавным и полезным. Но ограничился забавностью. Сценарий в примере 8.40 отображает кнопку, которая случайным образом меняет свою картинку при каждом нажатии.

Пример 8.40. PP4E\Gui\Tour\buttonpics-func.py

```
from tkinter import *          # импортировать базовый набор виджетов,
from glob import glob          # чтобы получить список файлов по расширению
import demoCheck               # прикрепить демонстрационный пример с флажками
import random                  # выбрать случайную картинку
gifdir = '../gifs/'            # каталог по умолчанию с GIF-файлами

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")                    # имеющиеся GIF-файлы
images = [(x, PhotoImage(file=x)) for x in files] # загрузить и сохранить
print(files)
root.mainloop()
```

В этом примере используются несколько встроенных инструментов из библиотеки Python:

- Модуль `glob`, с которым мы впервые встретились в главе 4, позволяет получить список всех файлов с расширением *.gif* в каталоге – иными словами, всех GIF-файлов, которые там хранятся.
- Модуль `random` используется для выбора случайного GIF-файла из числа имеющихся в каталоге: функция `random.choice` случайным образом выбирает и возвращает элемент из списка.
- Чтобы изменить отображаемое изображение (и имя GIF-файла в метке в верхней части окна), сценарий просто вызывает метод `config` виджета с новыми значениями параметров – такое изменение динамически изменяет вид графического элемента.

Для разнообразия этот сценарий также прикрепляет экземпляр демонстрационной панели флажков `demoCheck`, который, в свою очередь, прикрепляет экземпляр кнопки `Quitter`, написанной нами ранее в примере 8.7. Конечно, это искусственный пример, но он еще раз демонстрирует мощь классов компонентов.

Обратите внимание, что все изображения, создаваемые в этом сценарии, сохраняются в списке `images`. В данном случае генератор списков применяет вызов конструктора `PhotoImage` к каждому файлу *.gif* в каталоге с картинками и возвращает список кортежей `(filename, image-object)`, ссылка на который сохраняется в глобальной переменной (то же самое можно реализовать с помощью функции `map`, использующей `lambda`-функцию с одним аргументом). Напомню, что это убережет объекты изображений от утилизации сборщиком мусора в течение всего времени выполнения программы. На рис. 8.40 изображено окно этого сценария в Windows.

В этой черно-белой книге можно не заметить, что имя GIF-файла выводится красным по голубому фону метки в верхней части окна. Окно этой программы автоматически расширяется или сжимается, когда выводятся бóльшие или меньшие GIF-файлы. На рис. 8.41 показано изображение, случайно выбранное из каталога графических файлов, имеющее большую высоту.

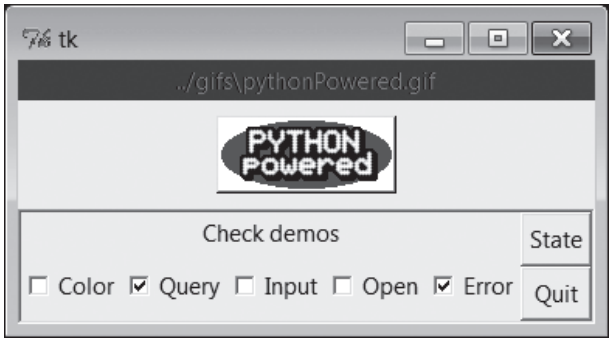


Рис. 8.40. *Сценарий buttonpics в действии*

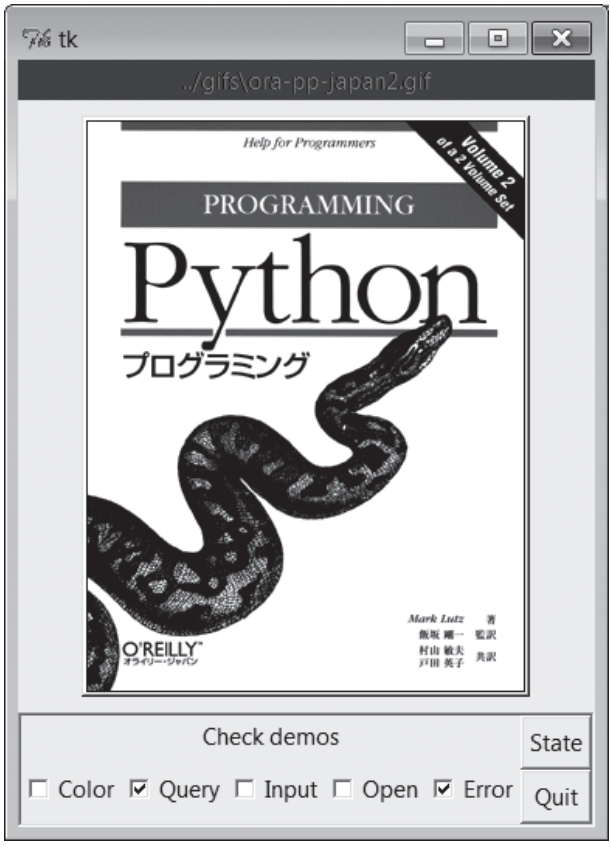


Рис. 8.41. *Сценарий buttonpics, отображающий более высокую картинку*

И наконец, на рис. 8.42 изображен графический интерфейс этого сценария, отображающий одну из более широких картинок в GIF-файле, выбранную совершенно случайно из каталога с картинками.[1]

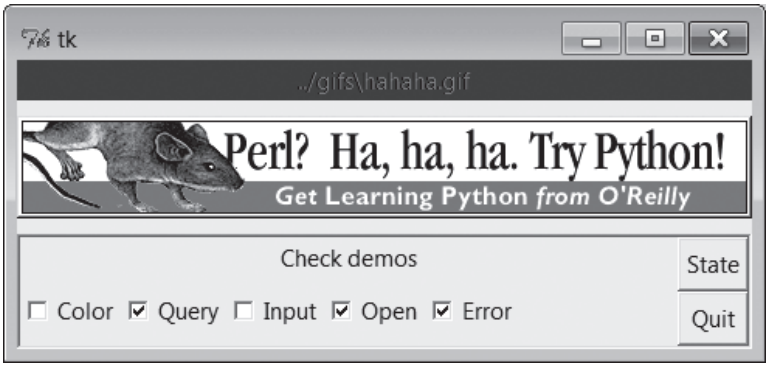


Рис. 8.42. *Сценарий buttonpics обретает политический подтекст*

В качестве продолжения наших развлечений перепишем этот сценарий в виде класса на случай, если когда-нибудь в будущем потребуется прикрепить или перекроить его (может произойти и такое, особенно в программах, имеющих практическую ценность). В основном такая переделка состоит в добавлении отступов и префикса `self` перед именами глобальных переменных, как показано в примере 8.41.

Пример 8.41. PP4E\Gui\Tour\buttonpics.py

```
from tkinter import *  # импортировать базовый набор виджетов,
from glob import glob  # чтобы получить список файлов по расширению
import demoCheck       # прикрепить демонстрационный пример с флажками
import random          # выбрать случайную картинку
gifdir = '../gifs/'    # каталог по умолчанию с GIF-файлами

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)
```

[1] Не я автор данной картинки – она появилась в качестве баннера на сайтах для разработчиков, таких как *slashdot.com*, когда в 1999 году вышло первое издание «Learning Python». Она породила такую волну возмущения приверженцев Perl, что издательство O'Reilly в конце концов было вынуждено убрать эту рекламу. Хотя возможно, именно поэтому она оказалась в этой книге.

```
            self.pix.pack(pady=10)
            demoCheck.Demo(self, relief=SUNKEN, bd=2).pack(fill=BOTH)
            files = glob(gifdir + "*.gif")
            self.images = [(x, PhotoImage(file=x)) for x in 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 действует точно так же, как и оригинал, но теперь ее можно прикрепить к любому другому графическому интерфейсу, в который вы пожелаете включить такую дурацкую кнопку.

Отображение и обработка изображений с помощью PIL

Как упоминалось ранее, сценарии на языке Python, использующие библиотеку tkinter, отображают изображения, связывая независимо созданные объекты изображений с действующими виджетами. На момент написания данной книги библиотека tkinter была способна отображать 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`.

Данное множество поддерживаемых форматов файлов ограничено лежащей в основе библиотекой Tk, а не самой библиотекой tkinter, и может быть расширено в будущем. Но если вам нужно сейчас вывести файлы в другом формате (например, в популярном формате JPEG), можно либо преобразовать файлы в один из поддерживаемых форматов с помощью программы обработки изображений, либо установить пакет расширения Python PIL, о котором говорилось в начале главы 7.

Пакет PIL, Python Imaging Library, – система, распространяемая с исходными текстами, поддерживает в данное время около 30 форматов графических файлов (в том числе GIF, JPEG, TIFF, PNG и BMP). В дополнение к расширению диапазона поддерживаемых форматов графических файлов пакет PIL также предоставляет инструменты для обработки изображений, включая геометрические преобразования, создание миниатюр, преобразование из одного формата в другой и многое другое.

Основы PIL

Чтобы воспользоваться инструментами из этого пакета, его сначала необходимо получить и установить: инструкции вы найдете на сайте *http://www.pythonware.com* (или поищите по строке «PIL», воспользо-

вавшись поисковой системой). Затем нужно просто использовать особые объекты PhotoImage и BitmapImage, импортируемые из модуля ImageTk пакета PIL, чтобы открывать файлы в других графических форматах. Это совместимая замена для одноименных классов из библиотеки tkinter, которую можно использовать везде, где tkinter предполагает использование объекта PhotoImage или BitmapImage (то есть в настройках объектов меток, кнопок, холстов, текстовых виджетов и меню).

Это означает, что стандартный программный код, использующий tkinter, как показано ниже:

```
from tkinter import *
imgobj = PhotoImage(file=imgdir + "spam.gif")
Button(image=imgobj).pack()
```

можно заменить таким программным кодом:

```
from tkinter import *
from PIL import ImageTk
photoimg = ImageTk.PhotoImage(file=imgdir + "spam.jpg")
Button(image=photoimg).pack()
```

или более многословным эквивалентом, который удобно использовать, когда сценарий не только отображает изображения, но и обрабатывает их:

```
from tkinter import *
from PIL import Image, ImageTk
imageobj = Image.open(imgdir + "spam.jpeg")
photoimg = ImageTk.PhotoImage(imageobj)
Button(image=photoimg).pack()
```

В действительности, чтобы задействовать пакет PIL для отображения изображений, достаточно лишь установить его и добавить в сценарий единственную инструкцию from, импортирующую альтернативный класс PhotoImage после загрузки оригинальной версии из tkinter. Остальной программный код можно оставить без изменений, но он будет способен отображать графические изображения в форматах JPEG, PNG и других:

```
from tkinter import *
from PIL.ImageTk import PhotoImage          # <== добавьте эту строку
imgobj = PhotoImage(file=imgdir + "spam.png")
Button(image=imgobj).pack()
```

Особенности установки пакета PIL зависят от платформы. В Windows достаточно лишь загрузить и запустить самоустанавливающийся файл. В результате пакет PIL будет сохранен в каталоге установки Python *Lib\site-packages*, а поскольку мастер установки автоматически добавит каталог пакета в путь поиска модулей, никаких дополнительных настроек путей не потребуется. Просто запустите мастер установки и затем импортируйте модули из пакета PIL. На других платформах вам может потребоваться распаковать загруженный архив с исходным

программным кодом и добавить путь к каталогу пакета PIL в начало переменной окружения PYTHONPATH – за дополнительной информацией обращайтесь на веб-сайт проекта PIL. (При работе над этим изданием книги я использовал предварительную версию PIL для Python 3.1, но к моменту, когда вы будете читать эти строки, должна быть выпущена официальная версия пакета.)

Рассказ о пакете PIL занял бы значительно больше места, чем может позволить эта книга. Например, он предоставляет инструменты для конвертирования изображений, изменения размеров и преобразования, причем некоторые из них можно использовать как самостоятельные программы командной строки, не имеющие непосредственного отношения к графическим интерфейсам. Пакет PIL наверняка станет стандартным компонентом вашего инструментального набора, особенно для ваших программ, выполняющих отображение или обработку изображений.

Дополнительную информацию вы найдете на сайте *http://www.pythonware.com*, а также в комплекте электронной документации по PIL и tkinter. Однако, чтобы помочь вам начать, мы закроем эту главу несколькими интересными примерами использования пакета PIL для отображения и обработки изображений.

Отображение других типов графических изображений с помощью PIL

В предыдущих примерах работы с изображениями мы прикрепляли виджеты к кнопкам и холстам, однако стандартный набор инструментов библиотеки tkinter позволяет прикреплять изображения к виджетам различных типов, включая простые метки, текстовые виджеты и элементы меню. Так, сценарий в примере 8.42 отображает изображение в метке, находящейся в главном окне приложения, используя только средства библиотеки tkinter. Этот сценарий предполагает, что изображения хранятся в подкаталоге *images*, а также позволяет передавать имя файла с изображением в аргументе командной строки (при отсутствии аргументов по умолчанию используется файл *spam.gif*). Кроме того, для большей переносимости он объединяет имена файлов и каталогов с помощью os.path.join и выводит высоту и ширину изображения в пикселях в стандартный поток вывода, исключительно чтобы предоставить дополнительную информацию.

Пример 8.42. PP4E\Gui\PIL\viewer-tk.py

```
"""
отображает изображение с помощью стандартного объекта PhotoImage из биб-
лиотеки tkinter; данная реализация может работать с GIF-файлами, но не может
обрабатывать изображения в формате JPEG; использует файл с изображением, имя
которого указано в командной строке, или файл по умолчанию; используйте Canvas
вместо Label, чтобы обеспечить возможность прокрутки, и т.д.
"""
```

```
import os, sys
from tkinter import * # использовать стандартный объект PhotoImage
                      # работает с форматом GIF, а для работы с форматом JPEG
                      # требуется пакет PIL
imgdir = 'images'
imgfile = 'london-2010.gif'
if len(sys.argv) > 1:                  # аргумент командной строки задан?
    imgfile = sys.argv[1]
imgpath = os.path.join(imgdir, imgfile)

win = Tk()
win.title(imgfile)
imgobj = PhotoImage(file=imgpath)
Label(win, image=imgobj).pack()        # прикрепить к метке Label
print(imgobj.width(), imgobj.height()) # вывести размеры в пикселях,
win.mainloop()                         # пока объект не уничтожен
```

На рис. 8.43 изображено окно этого сценария в Windows 7, где отображается изображение из GIF-файла по умолчанию. Запустите его из консоли, передав имя файла в виде аргумента командной строки, чтобы просмотреть другое изображение из подкаталога *images* (например, *python viewer_tk.py filename.gif*).

***Рис. 8.43.** Отображение картинки в формате GIF средствами tkinter*

Сценарий в примере 8.42 может работать только с изображениями, форматы которых поддерживаются базовым набором инструментов в библиотеке tkinter. Для отображения изображений в других форматах, таких как JPEG, необходимо установить пакет PIL и использовать его альтернативную реализацию класса `PhotoImage`. С точки зрения программного кода, для этого достаточно добавить всего одну инструкцию `import`, как показано в примере 8.43

Пример 8.43. PP4E\Gui\PIL\viewer-pil.py

```
"""
отображает изображение с помощью альтернативного объекта из пакета PIL
поддерживает множество форматов изображений; предварительно установите пакет
PIL: поместите его в каталог Lib\site-packages
"""

import os, sys
from tkinter import *
from PIL.ImageTk import PhotoImage # <== использовать альтернативный класс из
                                   # PIL, остальной программный код
                                   # без изменений
imgdir = 'images'
imgfile = 'florida-2009-1.jpg'     # поддерживает gif, jpg, png, tiff, и др.
if len(sys.argv) > 1:
    imgfile = sys.argv[1]
imgpath = os.path.join(imgdir, imgfile)

win = Tk()
win.title(imgfile)
imgobj = PhotoImage(file=imgpath)  # теперь поддерживает и JPEG!
Label(win, image=imgobj).pack()
win.mainloop()
print(imgobj.width(), imgobj.height()) # показать размер в пикселях при выходе
```

Задействовав пакет PIL, наш сценарий теперь способен отображать изображения в самых разных форматах, включая формат JPEG картинки по умолчанию, изображенной на рис. 8.44. Поэкспериментируйте со сценарием самостоятельно и попробуйте отобразить другие картинки.

Отображение всех изображений в каталоге

В рамках нашего примера не потребуется приложить много усилий, чтобы обеспечить возможность просмотра всех изображений в каталоге, опираясь на некоторые инструменты для работы с каталогами, с которыми мы познакомились в первой части книги. Сценарий в примере 8.44 просто открывает новое окно `Toplevel` для каждого файла изображения, обнаруженного в каталоге (указанном в в аргументе командной строки или используемом по умолчанию), пропуская файлы, не являющиеся изображениями, за счет обработки исключения, при этом сообщения об ошибках выводятся в окне, открытом для файла.

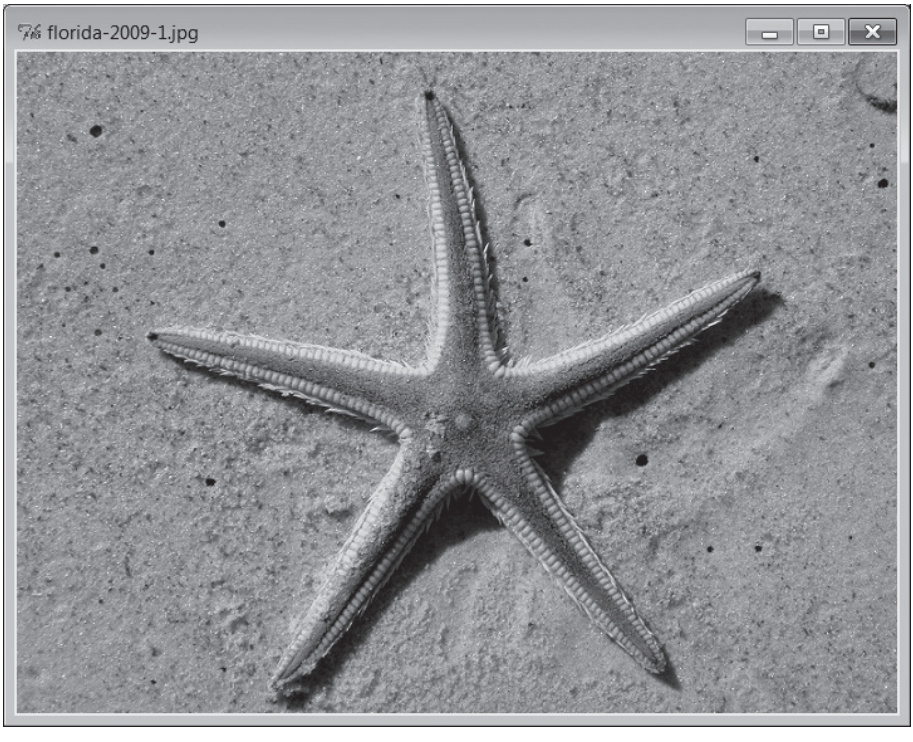


***Рис. 8.44.** Отображение картинки в формате GIF средствами tkinter и PIL*

Пример 8.44. PP4E\Gui\PIL\viewer-dir.py

```
"""
выводит все изображения, найденные в каталоге, открывая новые окна
GIF-файлы поддерживаются стандартными средствами  tkinter, но JPEG-файлы будут
пропускаться при отсутствии пакета PIL
"""

import os, sys
from tkinter import *
from PIL.ImageTk import PhotoImage # <== требуется для JPEG и др. форматов

imgdir = 'images'
if len(sys.argv) > 1: imgdir = sys.argv[1]
    imgfiles = os.listdir(imgdir)  # не включает полный путь к каталогу

main = Tk()
main.title('Viewer')
quit = Button(main, text='Quit all', command=main.quit, font=('courier', 25))
quit.pack()
savephotos = []
```

```
for imgfile in imgfiles:
    imgpath = os.path.join(imgdir, imgfile)
    win = Toplevel()
    win.title(imgfile)
    try:
        imgobj = PhotoImage(file=imgpath)
        Label(win, image=imgobj).pack()
        print(imgpath, imgobj.width(), imgobj.height()) # размер в пикселях
        savephotos.append(imgobj)                       # сохранить ссылку
    except:
        errmsg = 'skipping %s\n%s' % (imgfile, sys.exc_info()[1])
        Label(win, text=errmsg).pack()

main.mainloop()
```

Запустите этот сценарий у себя, чтобы посмотреть создаваемые им окна. При запуске он создает одно главное окно с кнопкой Quit, щелчок на которой закрывает все дополнительные окна, количество которых совпадает с количеством файлов изображений в каталоге. Этот сценарий удобно использовать для быстрой организации просмотра, но он определенно не является образцом дружественного отношения к пользователю, особенно если в каталоге содержится огромное количество изображений! Каталог *images*, находящийся в дереве примеров и использовавшийся при тестировании, содержит 59 изображений. В результате при просмотре этого каталога сценарий порождает 60 окон, а каталоги, где вы храните свои снимки, сделанные цифровой фотокамерой, могут содержать гораздо больше изображений. Чтобы улучшить сценарий, перейдем к следующему разделу.

Создание миниатюр изображений с помощью пакета PIL

Как уже упоминалось, пакет PIL позволяет не только отображать картинки в графическом интерфейсе, но и выполнять массу других операций над ними. В его составе имеются инструменты для изменения размеров изображений, преобразования из одного формата в другой и так далее. Один из таких инструментов предоставляет возможность генерировать «миниатюры» изображений из оригиналов. Такие миниатюры могут отображаться в веб-страницах или в инструментах выбора графического интерфейса, дающих пользователю возможность открывать полноразмерные изображения.

В примере 8.45 приводится конкретная реализация этой идеи – он генерирует миниатюры изображений, используя PIL, и отображает их на кнопках, которые открывают соответствующие оригинальные картинки в случае щелчка мышью. В результате получается графический интерфейс, очень напоминающий интерфейс проводников по файловой системе, ставший стандартом в современных операционных системах.

Но данный интерфейс реализован на языке Python, что дает нам полный контроль над его поведением и возможность повторного использования в наших приложениях. Фактически мы повторно будем использовать функцию makeThumbs, уже реализованную в других примерах. Как обычно, этот пример демонстрирует некоторые основные преимущества, характерные для открытого программного обеспечения.

Пример 8.45. PP4E\Gui\PIL\viewer_thumbs.py

```
"""
выводит все изображения, имеющиеся в каталоге, в виде миниатюр на кнопках,
щелчок на которых приводит к выводу полноразмерного изображения; требует
наличия пакета PIL для отображения JPEG-файлов и создания миниатюр; что сделать:
добавить прокрутку, если в окне выводится слишком много миниатюр!
"""
import os, sys, math
from tkinter import *
from PIL import Image # <== required for thumbs
from PIL.ImageTk import PhotoImage # <== required for JPEG display

def makeThumbs(imgdir, size=(100, 100), subdir='thumbs'):
    """
    создает миниатюры для всех изображений в каталоге; для каждого изображения
    создается и сохраняется новая миниатюра или загружается существующая;
    при необходимости создает каталог thumb;
    возвращает список кортежей (имя_файла_изображения, объект_миниатюры);
    для получения списка файлов миниатюр вызывающая программа может также
    воспользоваться функцией listdir в каталоге thumb; для неподдерживаемых
    типов файлов может возбуждать исключение IOError, или другое;
    ВНИМАНИЕ: можно было бы проверять время создания файлов;
    """
    thumbdir = os.path.join(imgdir, subdir)
    if not os.path.exists(thumbdir):
        os.mkdir(thumbdir)

    thumbs = []
    for imgfile in os.listdir(imgdir):
        thumbpath = os.path.join(thumbdir, imgfile)
        if os.path.exists(thumbpath):
            thumbobj = Image.open(thumbpath)     # использовать существующую
            thumbs.append((imgfile, thumbobj))
        else:
            print('making', thumbpath)
            imgpath = os.path.join(imgdir, imgfile)
            try:
                imgobj = Image.open(imgpath)     # создать новую миниатюру
                imgobj.thumbnail(size, Image.ANTIALIAS) # фильтр, дающий
                                                 # лучшее качество при
                                                 # уменьшении размеров
                imgobj.save(thumbpath)           # тип определяется расширением
```

```
                thumbs.append((imgfile, imgobj))
            except:                             # не всегда IOError
                print("Skipping: ", imgpath)
    return thumbs

class ViewOne(Toplevel):
    """
    открывает одно изображение в новом окне; ссылку на объект PhotoImage
    требуется сохранить: изображение будет утрачено при утилизации объекта;
    """
    def __init__(self, imgdir, imgfile):
        Toplevel.__init__(self)
        self.title(imgfile)
        imgpath = os.path.join(imgdir, imgfile)
        imgobj = PhotoImage(file=imgpath)
        Label(self, image=imgobj).pack()
        print(imgpath, imgobj.width(), imgobj.height())  # размер в пикселях
        self.savephoto = imgobj                          # сохранить ссылку
                                                         # на изображение
def viewer(imgdir, kind=Toplevel, cols=None):
    """
    создает окно с миниатюрами для каталога с изображениями: по одной кнопке с
    миниатюрой для каждого изображения;
    используйте параметр kind=Tk, чтобы вывести миниатюры в главном окне, или
    Frame (чтобы прикрепить к фрейму); значение imgfile изменяется в каждой
    итерации цикла: ссылка на значение должна сохраняться по умолчанию;
    объекты PhotoImage должны сохраняться: иначе при утилизации изображения
    будут уничтожены;
    компонует в ряды фреймов (в противоположность сеткам, фиксированным
    размерам, холстам);
    """
    win = kind()
    win.title('Viewer: ' + imgdir)
    quit = Button(win, text='Quit', command=win.quit, bg='beige') # добавить
    quit.pack(fill=X, side=BOTTOM)       # первой, чтобы урезалась последней
    thumbs = makeThumbs(imgdir)
    if not cols:
        cols = int(math.ceil(math.sqrt(len(thumbs)))) # фиксированное или N x N

    savephotos = []
    while thumbs:
        thumbsrow, thumbs = thumbs[:cols], thumbs[cols:]
        row = Frame(win)
        row.pack(fill=BOTH)
        for (imgfile, imgobj) in thumbsrow:
            photo   = PhotoImage(imgobj)
            link    = Button(row, image=photo)
            handler = lambda savefile=imgfile: ViewOne(imgdir, savefile)
            link.config(command=handler)
            link.pack(side=LEFT, expand=YES)
```

```
            savephotos.append(photo)
    return win, savephotos

if __name__ == '__main__':
    imgdir = (len(sys.argv) > 1 and sys.argv[1]) or 'images'
    main, save = viewer(imgdir, kind=Tk)
    main.mainloop()
```

Просьба обратить внимание, что функция `viewer` передает значение `imgfile` `lambda`-выражению, генерирующему обработчик, в виде *значения по умолчанию аргумента*. Это объясняется тем, что `imgfile` является переменной цикла, и если не сохранять текущее ее значение, все обработчики получат значение переменной, установленное в последней итерации (все кнопки будут открывать одно и то же изображение!). Отметьте также, что мы сохраняем ссылки на объекты изображений в списке – если этого не сделать, изображения будут *уничтожены* при утилизации их объектов сборщиком мусора, даже если в этот момент они будут отображаться на экране. Чтобы избежать такой неприятности, мы сохраняем ссылки на объекты в долгоживущем списке.

На рис. 8.45 изображено главное окно с миниатюрами, доступными для выбора, созданное сценарием из примера 8.45 при попытке просмотра содержимого каталога по умолчанию *images*, находящегося в дереве примеров (размер окна был изменен при подготовке снимка с экрана для книги). Как и прежде, вы можете передать сценарию имя другого каталога, чтобы просмотреть его содержимое (например, имя каталога, где вы храните свои цифровые фотографии). Щелчок на кнопке с миниатюрой в главном окне открывает соответствующее изображение в новом окне, как показано на рис. 8.46.

На данный момент большая часть примера 8.45 должна быть вам понятна. В нем кнопки с миниатюрами прикрепляются к *рядам фреймов*, как и в предыдущих примерах (смотрите альтернативные варианты компоновки форм ввода, реализованные выше в этой главе). Основная часть программного кода, использующего пакет PIL, находится в функции `makeThumbs`. Он открывает, создает и сохраняет изображения миниатюр, если ранее они не были сохранены (то есть не были кэшированы) в локальных файлах. В данной реализации миниатюры сохраняются в том же формате, что и оригинальные, полноразмерные изображения.

Мы также использовали фильтр PIL `ANTIALIAS`, дающий наилучшее качество при уменьшении размеров изображения, что особенно заметно для изображений в формате GIF с низким разрешением. Создание миниатюры сводится, по сути, к изменению размеров с сохранением оригинальных пропорций. Из-за невозможности охватить все особенности здесь за дополнительными подробностями, касающимися API, я отсылаю вас к пакету PIL и документации по нему.

В следующей главе мы еще раз коротко вернемся к проблеме создания миниатюр, когда будем создавать кнопки для панели инструментов.

Рис. 8.45. *Простой графический интерфейс выбора миниатюр, простые ряды фреймов*

Рис. 8.46. *Окно с полноразмерным изображением*

Однако, прежде чем двинуться дальше, быстро познакомимся с тремя вариантами реализации отображения миниатюр – в первом из них делается акцент на оценке производительности, а в других двух будет улучшена, вероятно, не самая удачная компоновка миниатюр, изображенная на рис. 8.45.

Производительность: сохранение миниатюр в файлах

В текущей реализации сценарий сохраняет созданные миниатюры в файлах, благодаря чему при следующем запуске они могут быть загружены значительно быстрее. Строго говоря, делать это совсем не обязательно – в примере 8.46 реализована измененная версия функции создания миниатюр, которая также генерирует изображения миниатюр в памяти, но не сохраняет их.

Для небольших коллекций изображений разница в скорости выполнения будет незаметна. Однако если испытать эти альтернативы на больших коллекциях изображений, можно будет заметить, что оригинальная версия в примере 8.45, сохраняющая и загружающая миниатюры из файлов, дает значительное преимущество в скорости. В одном из тестов с большой коллекцией файлов изображений на моем компьютере (примерно 320 цифровых фотографий на весьма, по общему мнению, медлительном ноутбуке) оригинальному сценарию потребовалось всего 5 секунд, чтобы открыть графический интерфейс (после первого запуска, в ходе которого было выполнено кэширование миниатюр), тогда как версии, представленной в примере 8.46, потребовалась 1 минута и 20 секунд: в 16 раз больше. Загрузка миниатюр из файлов выполняется значительно быстрее, чем операция изменения размеров.

Пример 8.46. PP4E\Gui\PIL\viewer-thumbs-nosave.py

```
"""
то же самое, но не сохраняет и не загружает миниатюры из файлов:
для маленьких коллекций изображений, кажется, работает также быстро, но для
больших коллекций, при сохранении миниатюр в файлах, запуск происходит намного
быстрее; в некоторых приложениях (например, в веб-страницах) сохранение может
оказаться насущной необходимостью
"""
import os, sys
from PIL import Image
from tkinter import Tk
import viewer_thumbs

def makeThumbs(imgdir, size=(100, 100), subdir='thumbs'):
    """
    создает миниатюры в памяти, но не сохраняет их в файлах
    """
    thumbs = []
    for imgfile in os.listdir(imgdir):
        imgpath = os.path.join(imgdir, imgfile)
```

```
            try:
                imgobj = Image.open(imgpath)       # создать новую миниатюру
                imgobj.thumbnail(size)
                thumbs.append((imgfile, imgobj))
            except:
                print("Skipping: ", imgpath)
        return thumbs

if __name__ == '__main__':
    imgdir = (len(sys.argv) > 1 and sys.argv[1]) or 'images'
    viewer_thumbs.makeThumbs = makeThumbs
    main, save = viewer_thumbs.viewer(imgdir, kind=Tk)
    main.mainloop()
```

Варианты компоновки: по сетке

Следующая версия нашего сценария содержит изменения, носящие исключительно косметический характер, но она демонстрирует некоторые важные понятия компоновки tkinter. Если посмотреть на рис. 8.45 достаточно пристально, можно заметить, что миниатюры располагаются недостаточно однородно. Если расположение отдельных строк еще как-то согласуется благодаря использованию рядов фреймов, то стройность колонок может нарушаться из-за разных размеров изображений. Применение различных параметров компоновки, похоже, не способно исправить этот недостаток (и может даже ухудшить положение дел – попробуйте сами), а компоновка по фреймам-колонкам просто переместит проблему в другую размерность. При работе с крупными коллекциями это может осложнить поиск какого-то определенного изображения.

Приложив совсем немного усилий, мы можем добиться более стройного размещения миниатюр, либо разместив их по сетке, либо использовав кнопки фиксированного размера. Сценарий в примере 8.47 располагает кнопки по сетке, используя менеджер компоновки `grid` – более подробно мы будем рассматривать его в следующей главе. Как и представление холстов выше, некоторые фрагменты этого примера следует рассматривать, как предварительное знакомство. В двух словах, функция `grid` располагает содержимое по рядам и колонкам – все приемы, обеспечивающие неподвижность кнопки Quit, будут рассматриваться в главе 9.

Пример 8.47. PP4E\Gui\PIL\viewer-thumbs-grid.py

```
"""
то же, что и viewer_thumbs, но использует менеджер компоновки grid, чтобы
добиться более стройного размещения миниатюр; того же эффекта можно добиться
с применением фреймов и менеджера pack, если кнопки будут иметь фиксированный
и одинаковый размер;
"""
import sys, math
from tkinter import *
from PIL.ImageTk import PhotoImage
```

```
from viewer_thumbs import makeThumbs, ViewOne

def viewer(imgdir, kind=Toplevel, cols=None):
    """
    измененная версия, размещает миниатюры по сетке
    """
    win = kind()
    win.title('Viewer: ' + imgdir)
    thumbs = makeThumbs(imgdir)
    if not cols:
        cols = int(math.ceil(math.sqrt(len(thumbs))))# фиксированное или N x N

    rownum = 0
    savephotos = []
    while thumbs:
        thumbsrow, thumbs = thumbs[:cols], thumbs[cols:]
        colnum = 0
        for (imgfile, imgobj) in thumbsrow:
            photo   = PhotoImage(imgobj)
            link    = Button(win, image=photo)
            handler = lambda savefile=imgfile: ViewOne(imgdir, savefile)
            link.config(command=handler)
            link.grid(row=rownum, column=colnum)
            savephotos.append(photo)
            colnum += 1
        rownum += 1

    Button(win, text='Quit', command=win.quit).grid(columnspan=cols, stick=EW)
    return win, savephotos

if __name__ == '__main__':
    imgdir = (len(sys.argv) > 1 and sys.argv[1]) or 'images'
    main, save = viewer(imgdir, kind=Tk)
    main.mainloop()
```

Эффект размещения по сетке изображен на рис. 8.47 – наши кнопки образовали более стройные ряды и колонки, чем на рис. 8.45, потому что они позиционируются не только по рядам, но и по колонкам. Как мы увидим в следующей главе, размещение по сетке можно использовать всякий раз, когда схема размещения элементов интерфейса носит двухмерный характер.

Варианты компоновки: кнопки фиксированного размера

Компоновка по сетке улучшает картину, – картинки располагаются ровными рядами и колонками – но сама форма картинок оставляет желать лучшего. Мы могли бы добиться еще более стройного размещения, использовав кнопки фиксированного размера. По умолчанию размеры кнопок автоматически подстраиваются под размер изображения (или

текста), но мы всегда можем зафиксировать размер кнопок. Эта уловка используется в примере 8.48. Данный сценарий устанавливает высоту и ширину каждой кнопки в соответствии с максимальным размером миниатюр, чтобы ни одна кнопка не оказалась ни слишком узкой, ни слишком низкой. Если для всех миниатюр будет установлен один и тот же максимальный размер (что гарантируется нашей функцией создания миниатюр), это обеспечит желаемое размещение кнопок.

Рис. 8.47. *Графический интерфейс выбора миниатюр с размещением по сетке*

Пример 8.48. PP4E\Gui\PIL\viewer-thumbs-fixed.py

```
"""
использует кнопки фиксированного размера для миниатюр, благодаря чему
достигается еще более стройное размещение; размеры определяются по объектам
изображений, при этом предполагается, что для всех миниатюр был установлен один
и тот же максимальный размер; по сути именно это и делают графические интерфейсы
файловых менеджеров;
"""

import sys, math
from tkinter import *
from PIL.ImageTk import PhotoImage
from viewer_thumbs import makeThumbs, ViewOne

def viewer(imgdir, kind=Toplevel, cols=None):
```

```
    """
    измененная версия, выполняет размещение с использованием кнопок
    фиксированного размера
    """
    win = kind()
    win.title('Viewer: ' + imgdir)
    thumbs = makeThumbs(imgdir)
    if not cols:
        cols = int(math.ceil(math.sqrt(len(thumbs))))# фиксированное или N x N

    savephotos = []
    while thumbs:
        thumbsrow, thumbs = thumbs[:cols], thumbs[cols:]
        row = Frame(win)
        row.pack(fill=BOTH)
        for (imgfile, imgobj) in thumbsrow:
            size = max(imgobj.size)            # ширина, высота
            photo = PhotoImage(imgobj)
            link = Button(row, image=photo)
            handler = lambda savefile=imgfile: ViewOne(imgdir, savefile)
            link.config(command=handler, width=size, height=size)
            link.pack(side=LEFT, expand=YES)
            savephotos.append(photo)

    Button(win, text='Quit', command=win.quit, bg='beige').pack(fill=X)
    return win, savephotos

if __name__ == '__main__':
    imgdir = (len(sys.argv) > 1 and sys.argv[1]) or 'images'
    main, save = viewer(imgdir, kind=Tk)
    main.mainloop()
```

Результат применения кнопок фиксированного размера изображен на рис. 8.48 – теперь все кнопки имеют одинаковые размеры, вычисленные на основании размеров имеющихся изображений. Все миниатюры отображаются как одинаковые элементы мозаики независимо от их формы, что упрощает и просмотр. Естественно, возможны и другие схемы размещения – поэкспериментируйте с некоторыми параметрами настройки в этом сценарии, чтобы посмотреть их воздействие на графический интерфейс.

Прокрутка и холсты (забегая вперед)

Сценарий отображения миниатюр, представленный в этом разделе, неплохо справляется со своей задачей при умеренном количестве изображений в каталогах; для просмотра крупных коллекций изображений можно также задать настройки с уменьшенными размерами миниатюр. Но, вероятно, самое большое ограничение этих программ состоит в том, что для каталогов с очень большим количеством изображений окна бу-

Рис. 8.48. *Графический интерфейс выбора миниатюр с кнопками фиксированного размера и рядами фреймов*

дут получаться слишком большими и неудобными в работе (при этом часть миниатюр вообще может не поместиться в окне).

Даже при просмотре каталога с изображениями из дерева примеров для этой книги мы потеряли кнопку Quit в нижней части окна, как показано на двух последних рисунках, потому что миниатюр оказалось слишком много, чтобы уместить их в этом окне. В качестве иллюстрации различий посмотрите оригинальную версию в примере 8.45: там кнопка Quit добавляется первой именно поэтому – чтобы ее урезание происходило в последнюю очередь, после всех миниатюр, и поэтому она остается видимой даже при слишком большом количестве картинок. Мы могли бы применить тот же прием и в других версиях, но при большом количестве миниатюр они все равно не поместились бы все в окно. При просмотре каталогов с огромным количеством цифровых фотографий может получиться окно, слишком большое, чтобы поместиться на экране компьютера.

Чтобы добиться большего успеха, можно было бы поместить миниатюры на виджет, поддерживающий возможность прокрутки. Свободно распространяемый пакет Pmw предоставляет удобный фрейм с прокруткой, который можно было бы использовать. Кроме того, в библиотеке tkinter имеется стандартный виджет Canvas, обеспечивающий возможность более точного управления параметрами отображения картинок (включая

их размещение по абсолютным координатам в пикселях) и поддерживающий горизонтальную и вертикальную прокрутку своего содержимого.

В следующей главе мы напишем последнюю версию этого примера, которая реализует все эти идеи, – она отображает миниатюры в холсте с прокруткой и потому лучше подходит для работы с большими коллекциями изображений. В ней используются кнопки фиксированного размера, как и в последнем примере здесь, но позиционирование их выполняется по вычисленным координатам. Я не буду обсуждать здесь дальнейшие детали, потому что новую версию мы будем рассматривать в соединении с холстами в следующей главе. А в главе 11 мы используем этот прием в еще более полноценной программе для работы с изображениями, которая называется PyPhoto.

Чтобы узнать, как действуют эти программы, необходимо перейти к следующей главе – второй части нашего тура по виджетам.

9

Экскурсия по tkinter, часть 2

«Меню дня: Spam, Spam и еще раз Spam»

Это вторая глава обзора библиотеки tkinter, состоящего из двух частей. Она продолжает движение с того места, на котором остановилась глава 8, и рассказывает о некоторых более сложных виджетах и инструментах из арсенала tkinter. В этой главе представлены следующие темы:

- Виджеты `Menu`, `Menubutton` и `OptionMenu`
- Виджет `Scrollbar`: для прокрутки текста, списков и холстов
- Виджет `Listbox`: список с возможностью выбора нескольких вариантов
- Виджет `Text`: универсальный инструмент отображения и редактирования текста
- Виджет `Canvas`: универсальный инструмент вывода графики
- Менеджер компоновки `grid`, принцип действия которого основан на использовании таблиц
- Инструменты измерения интервалов времени: `after`, `update`, `wait` и потоки выполнения
- Основы анимации в tkinter
- Буферы обмена, удаление виджетов и окон и так далее.

К концу 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 главах 10 и 11. А сейчас продолжим обзор виджетов.

Меню

Меню представляют собой раскрывающиеся списки, которые обычно можно увидеть в верхней части окна (или всего экрана, если вы работаете на Macintosh). Переместите указатель мыши на панель меню, щелкните на имени (например, Файл (File)), и под этим именем появится список вариантов выбора (например, Открыть (Open), Сохранить (Save)). Пункты меню могут запускать какие-либо действия, как щелчок на кнопке. Они могут также открывать другие «каскадные» подменю, выводящие дополнительные списки вариантов, показывать окна диалогов и так далее. В библиотеке 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 использоваться в качестве вложенных компонентов.

Меню окон верхнего уровня

Во всех последних версиях Python (где используется библиотека Tk версии 8.0 и выше) можно связывать горизонтальную панель меню с объектом окна верхнего уровня (например, `Tk` или `Toplevel`). В Windows и Unix (X Window) эта строка меню выводится вдоль верхнего края окна. В некоторых версиях Mac OS это меню при выборе окна заменяет то, что отображается 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 виджетов `Menu`. Нужно просто связать с окном один элемент `Menu` верхнего уровня, добавить другие объекты раскрывающихся меню в качестве каскадов для меню верхнего уровня и добавить элементы в каждый из раскрывающихся списков. Виджеты `Menu` перекрестно связываются со следующим, более высоким уровнем, с помощью аргумента родительского виджета и метода `add_cascade` виджета `Menu`. Делается это так:

1. Создать меню `Menu` верхнего уровня как дочерний элемент окна и записать ссылку на новый виджет `Menu` в атрибут `menu`.
2. Для каждого раскрывающегося меню создать новый объект `Menu` как дочерний для самого верхнего меню и добавить его как каскадный для самого верхнего меню с помощью метода `add_cascade`.
3. В каждое раскрывающееся меню, созданное на шаге 2, добавить элементы выбора вызовом метода `add_comand`, которому в аргументе `command` передать обработчик события выбора этого элемента.
4. Добавить каскадные подменю, создавая новые виджеты `Menu` как дочерние для того объекта `Menu`, который должен расширяться каскадно, и связывая родительский и дочерний объекты с помощью метода `add_cascade`.

В конечном итоге получится дерево виджетов Menu с ассоциированными обработчиками событий. Однако все это, вероятно, проще показать в программном коде, чем на словах. В примере 9.1 создается главное меню с двумя раскрывающимися меню, File и Edit; в раскрывающемся меню Edit имеется собственное вложенное подменю.

Пример 9.1. PP4E\Gui\Tour\menu_win.py

```
# меню окна верхнего уровня в стиле Tk8.0

from tkinter import *                   # импортировать базовый набор виджетов
from tkinter.messagebox import *        # импортировать 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=False)
    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=True)
    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')              # информация для менеджера окон
    makemenu(root)                      # создать строку 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()
```

Значительная часть программного кода в этом примере выполняет установку обработчиков событий и тому подобного, поэтому полезно будет выделить фрагменты, участвующие в процессе построения дерева меню. Для меню File:

```
top = Menu(win)                          # прикрепить Menu к окну
win.config(menu=top)                     # связать окно и меню
file = Menu(top)                         # прикрепить Menu к Menu верх. ур.
top.add_cascade(label='File', menu=file) # связать родителя с потомком
```

Помимо построения дерева объектов меню этот сценарий демонстрирует некоторые часто встречающиеся параметры конфигурации меню:

Линии-разделители

С помощью метода `add_separator` сценарий создает в меню Edit разделитель – это просто линия, используемая для разделения групп родственных пунктов.

Линии отрыва

Сценарий запрещает отрыв раскрывающегося меню Edit, передавая параметр `tearoff=0` при создании виджета `Menu`. Линии отрыва – это пунктирные линии, по умолчанию появляющиеся в меню tkinter верхнего уровня. Щелчок на этой линии создает новое окно, содержащее меню. Они могут служить удобным средством упрощения навигации (можно сразу щелкнуть на пункте отрывного меню, не блуждая по дереву вложенных пунктов), но не на всех платформах принято их использовать.

Горячие клавиши

Сценарий использует параметр `underline`, чтобы назначить уникальную букву в пункте меню горячей клавишей. Параметр задает смещение буквы в строке метки пункта меню. Например, в Windows пункт Quit в меню File этого сценария можно выбрать, как обычно, с помощью мыши, а также нажатием клавиши Alt, затем f и затем q. Использовать параметр `underline` не обязательно – в Windows первая буква имени раскрывающегося меню автоматически становится горячей клавишей, а кроме того, для перемещения по меню и выбора раскрывающихся пунктов можно использовать клавиши со стрелками и Enter. Но явное определение горячих клавиш может облегчить использование больших меню. Например, последовательность клавиш Alt+E+S+S выполняет действие по завершению программы, предусмотренное пунктом Spam во вложенном подменю Stuff, без каких-либо перемещений с помощью мыши или клавиш со стрелками.

Посмотрим, во что все это превращается в переводе на пиксели. На рис. 9.1 изображено окно, появляющееся при запуске этого сценария в Windows 7 с моими настройками системы. Оно несколько иначе, но похоже отображается в Unix и Macintosh.

На рис. 9.2 изображено, что происходит при выборе раскрывающегося меню File. Обратите внимание, что виджеты `Menu` связываются, а не присоединяются с помощью менеджера компоновки – в действительности менеджер компоновки здесь вообще не участвует. Если запустить этот

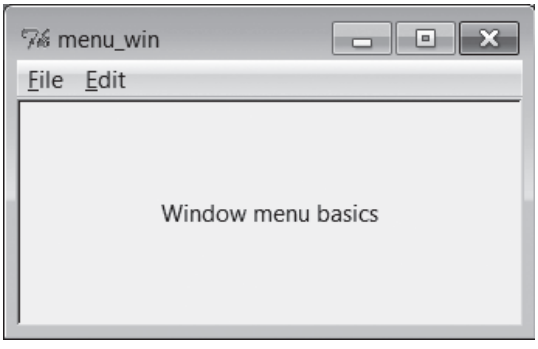


Рис. 9.1. *Сценарий menu_win: строка меню окна верхнего уровня*

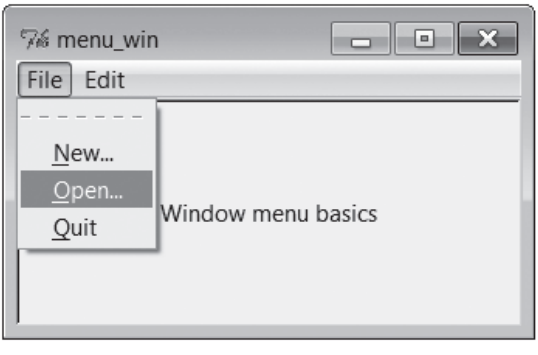


Рис. 9.2. *Открытое меню File*

сценарий, можно также заметить, что все элементы меню либо завершают выполнение программы, либо выводят стандартный диалог ошибки «Not Implemented» (Не реализовано). Этот пример просто демонстрирует работу с меню, но на практике обработчики событий выбора пунктов меню обычно выполняют более полезные вещи.

И наконец, на рис. 9.3 изображено, что происходит после щелчка по линии отрыва в меню File и выбора каскадного подменю в раскрывающемся меню Edit. Каскадные меню можно вкладывать друг в друга на любую глубину (но злоупотребление этой возможностью может запутать пользователей).

В библиотеке tkinter любое окно верхнего уровня может иметь собственное меню, в том числе и всплывающие окна, создаваемые виджетом `Toplevel`. Сценарий в примере 9.2 создает три всплывающих окна с такой же панелью меню, как в предыдущем примере. Если запустить его, он создаст картину, изображенную на рис. 9.4.

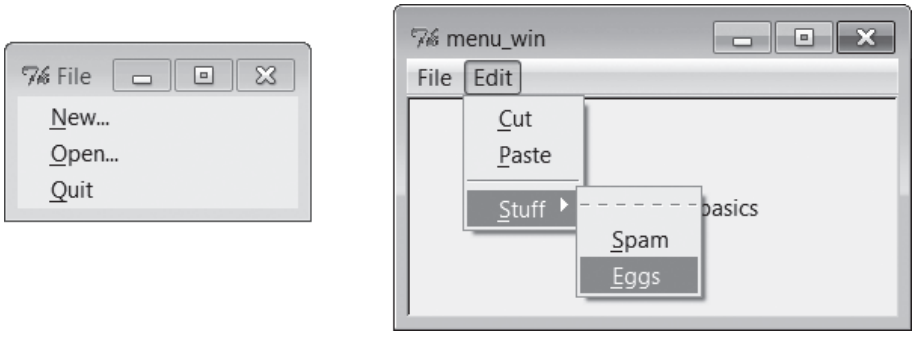


Рис. 9.3. *Отрывное меню File и каскадное Edit*

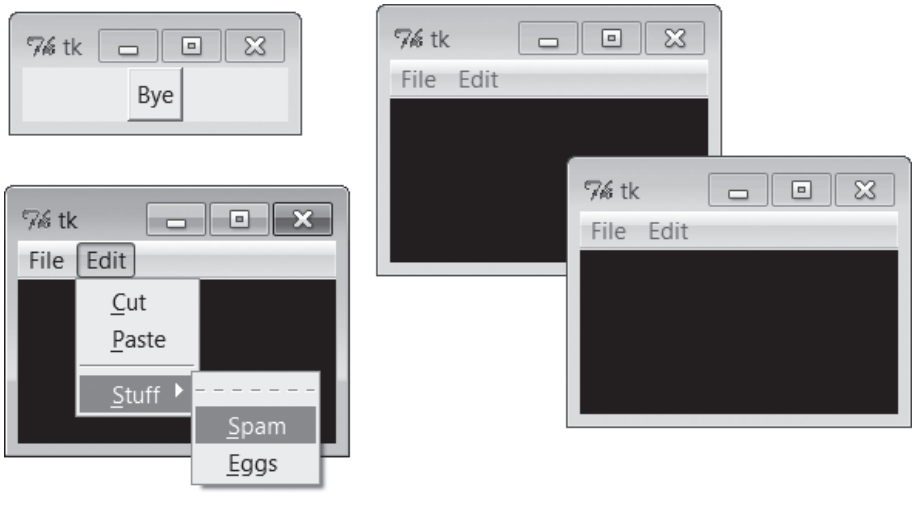


Рис. 9.4. *Несколько окон верхнего уровня с меню*

Пример 9.2. PP4E\Gui\Tour\menu_win-multi.py

```
from menu_win import makemenu # повторно использовать функцию создания меню
from tkinter import *

root = Tk()
for i in range(3):            # три всплывающих окна с меню
    win = Toplevel(root)
    makemenu(win)
    Label(win, bg='black', height=5, width=25).pack(expand=YES, fill=BOTH)
Button(root, text="Bye", command=root.quit).pack()
root.mainloop()
```

Меню на основе виджетов Frame и Menubutton

Хотя это не совсем обычно для окон верхнего уровня, но допустимо создание строки меню в виде горизонтального фрейма Frame. Однако прежде чем показывать, как это делается, я хочу объяснить, зачем это может понадобиться. Поскольку такая схема, основанная на фреймах, не зависит от протоколов окон верхнего уровня, она может применяться для добавления меню в качестве вложенных компонентов более крупных интерфейсов. Иными словами, она в основном применяется не к окнам верхнего уровня. Например, текстовый редактор PyEdit из главы 11 может использоваться как программа и как прикрепляемый компонент. Мы реализуем выбор в PyEdit с помощью оконных меню при выполнении его как самостоятельной программы и с помощью меню, основанного на фрейме, когда PyEdit будет встраиваться в интерфейсы PyMailGUI и PyView. Поэтому стоит знать обе схемы.

Для меню на основе фреймов требуется написать несколько дополнительных строчек программного кода, но они ненамного сложнее оконных меню. Для создания такого меню нужно разместить виджеты Menubutton в контейнере Frame, связать виджеты Menu и Menubutton и присоединить Frame к верхней части окна-контейнера. В примере 9.3 создается такое же меню, как в примере 9.2, но с использованием фрейма.

Пример 9.3. PP4E\Gui\Tour\menu_frm.py

```
# Меню на основе фреймов: пригодно для окон верхнего уровня и компонентов

from tkinter import *                  # импортировать базовый набор виджетов
from tkinter.messagebox import *       # импортировать 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=False)
    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=True)
    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)                       # создать строку 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 меню File она сводится к следующему:

```
menubar = Frame(parent)                    # создать Frame для строки меню
fbutton = Menubutton(menubar, text='File') # прикрепить Menubutton к Frame
file = Menu(fbutton)                       # прикрепить Menu к Menubutton
fbutton.config(menu=file)                  # связать кнопку и меню
```

В этой схеме появился дополнительный виджет `Menubutton`, но это не сделало ее намного сложнее создания оконных меню верхнего уровня. На рис. 9.5 и 9.6 изображено, как этот сценарий выполняется в Windows.

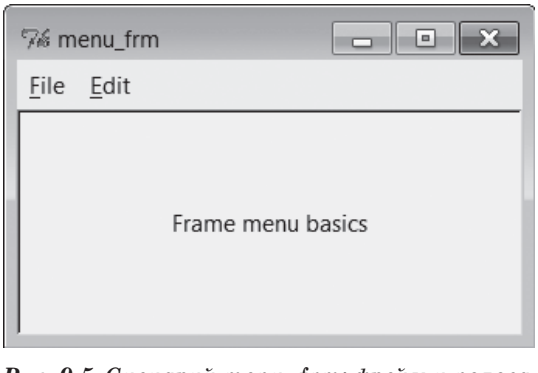


Рис. 9.5. *Сценарий menu_frm: фрейм и полоса меню Menubutton*

Виджеты меню в этом сценарии по умолчанию предоставляют набор связанных событий, автоматически отображающих меню при выборе мышью. Такие внешний вид и поведение не вполне совпадают с продемонстрированной выше схемой оконного меню верхнего уровня, хотя

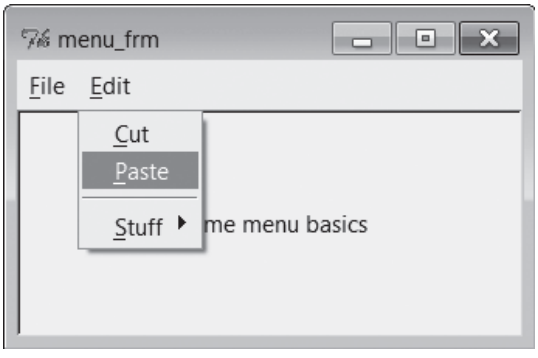


Рис. 9.6. *С выбранным меню Edit*

и близки к ней; виджеты могут настраиваться теми способами, которые допускают фреймы (то есть с помощью цветов и границ), и будут иметь одинаковый внешний вид на всех платформах (что, возможно, является недостатком).

Однако самым большим преимуществом строк меню на основе фреймов является возможность прикрепления в качестве вложенных компонентов к более крупным интерфейсам. Пример 9.4 и создаваемый им интерфейс (рис. 9.7) показывают, каким образом это делается, – обе полосы меню в одном окне полностью функциональны.

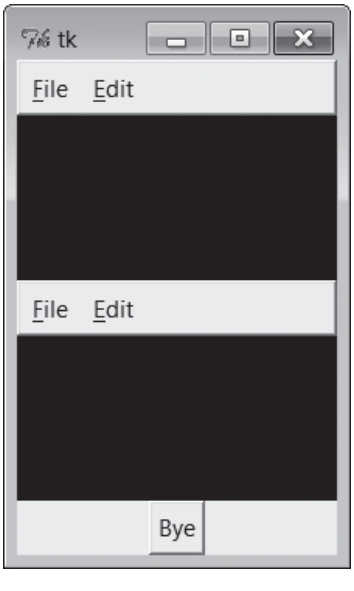


Рис. 9.7. *Несколько меню на основе фреймов в одном окне*

Пример 9.4. PP4E\Gui\Tour\menu_frm-multi.py

```
from menu_frm import makemenu  # здесь нельзя использовать menu_win--одно окно
from tkinter import *          # но можно прикреплять меню на основе фреймов

root = Tk()
for i in range(2):             # 2 меню в одном окне
    mnu = makemenu(root)
    mnu.config(bd=2, relief=RAISED)
    Label(root, bg='black', height=5, width=25).pack(expand=YES, fill=BOTH)
Button(root, text="Bye", command=root.quit).pack()
root.mainloop()
```

Благодаря отсутствию привязки к охватывающему окну меню на основе фреймов можно использовать в составе другого прикрепляемого компонента. Например, прием встраивания меню, представленный в примере 9.5, действует, даже когда родителем меню является другой контейнер `Frame`, а не окно верхнего уровня. Этот сценарий похож на предыдущий, но он создает три полнофункциональных строки меню, прикрепленных к фреймам внутри окна.

Пример 9.5. PP4E\Gui\Tour\menu_frm-multi2.py

```
from menu_frm import makemenu  # нельзя использовать menu_win--корень=Frame
from tkinter import *
root = Tk()
for i in range(3):             # три 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=25).pack(expand=YES, fill=BOTH)
Button(root, text="Bye", command=root.quit).pack()
root.mainloop()
```

Использование виджетов Menubutton и Optionmenu

В действительности меню, основанные на `Menubutton`, являются еще более универсальными, чем следует из примера 9.3, – они могут появляться в любом месте интерфейса, где может располагаться обычная кнопка, а не только в строке меню во фрейме `Frame`. В примере 9.6 создается раскрывающийся список `Menubutton`, который отображается самостоятельно и прикреплен к корневому окну. На рис. 9.8 приведен графический интерфейс, создаваемый этим примером.

Пример 9.6. PP4E\Gui\Tour\mbutton.py

```
from tkinter import *
root    = Tk()
mbutton = Menubutton(root, text='Food') # отдельное раскрывающееся меню
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()
```

В библиотеке tkinter имеется родственный виджет `Optionmenu`, который отображает выбранный элемент раскрывающегося меню. Он похож на `Menubutton`, к которому добавлена метка, и при щелчке на нем выводит меню вариантов выбора. Однако, чтобы получить результат выбора, нужно не регистрировать обработчик, а связывать переменные tkinter (описанные в главе 8), при этом элементы меню передаются конструктору в виде аргументов вслед за переменной.

***Рис. 9.8.** Виджет Menubutton сам по себе*

Пример 9.7 иллюстрирует типичное использование виджета `Optionmenu` и создает интерфейс, изображенный на рис. 9.9. Щелчок на любой из первых двух кнопок открывает раскрывающееся меню. Щелчок на третьей кнопке state выводит текущие значения, отображаемые на первых двух.

Пример 9.7. PP4E\Gui\Tour\optionmenu.py

```
from tkinter import *
root = Tk()

var1 = StringVar()
var2 = StringVar()
opt1 = OptionMenu(root, var1, 'spam', 'eggs', 'toast')   # как и Menubutton,
opt2 = OptionMenu(root, var2, 'ham', 'bacon', 'sausage') # но отображает
opt1.pack(fill=X)                                        # выбранный вариант
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()
```

Рис. 9.9. *Виджет Optionmenu в действии*

Есть и другие относящиеся к меню темы, которые мы пропустим в интересах экономии места. Например, сценарии могут добавлять элементы в системные меню и создавать всплывающие меню (отображаемые в ответ на события, не будучи связанными с кнопкой). Подробности по этой теме вы найдете в ресурсах Tk и tkinter.

Кроме простых пунктов и каскадов, меню могут также содержать неактивные пункты, флажки и переключатели, графические изображения. В следующем разделе демонстрируется, как программируются некоторые из этих специальных пунктов меню.

Окна с меню и панелью инструментов

Помимо меню, отображаемого в верхней части, окна часто содержат ряд кнопок в нижней части. Этот нижний ряд кнопок обычно называют панелью инструментов, и он нередко содержит кнопки для выполнения наиболее часто используемых операций, присутствующих в главном меню. С помощью библиотеки tkinter добавить в окно панель инструментов достаточно просто: нужно прикрепить кнопки (и другие виджеты) к фрейму, прикрепить фрейм к нижней границе окна и определить для него возможность растягивания только в горизонтальном направлении. Это просто очередная реализация иерархической структуры графического интерфейса, однако нужно следить, чтобы панели инструментов (и строки меню, основанные на фреймах) прикреплялись раньше других виджетов, чтобы при сжатии окна сначала обрезались виджеты, находящиеся в середине экрана, – обычно бывает желательно, чтобы панели инструментов и полосы меню имели приоритет перед другими виджетами.

В примере 9.8 представлен один из способов добавления панели инструментов в окно. Он демонстрирует также, как добавлять изображения в пункты меню (присвоить атрибуту `image` ссылку на объект `PhotoImage`) и как делать пункты меню недоступными для выбора, изображая их в серых тонах (вызвать метод меню `entryconfig`, передав ему индекс отключаемого пункта; отсчет начинается с 1). Обратите внимание, что объекты `PhotoImage` сохраняются в виде списка – напомню, что в отличие от других виджетов, они будут утеряны, если не сохранить ссылки на них (загляните в главу 8, если вам требуется освежить память).

Пример 9.8. PP4E\Gui\Tour\menuDemo.py

```
#!/usr/local/bin/python
"""
главное меню окна в стиле Tk8.0
строка меню и панель инструментов прикрепляются к окну в первую очередь, fill=X
(прикрепить первым = обрезать последним); добавляет изображения в элементы меню;
смотрите также: add_checkbutton, add_radiobutton
"""

from tkinter import *                  # импортировать базовый набор виджетов
from tkinter.messagebox import *       # импортировать стандартные диалоги

class NewMenuDemo(Frame):              # расширенный фрейм
    def __init__(self, parent=None):   # прикрепляется к корневому окну?
        Frame.__init__(self, parent)   # вызвать метод суперкласса
        self.pack(expand=YES, fill=BOTH)
        self.createWidgets()           # прикрепить фреймы/виджеты
        self.master.title("Toolbars and Menus")  # для менеджера окон
        self.master.iconname("tkpython")         # текст метки 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=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 = ('ora-lp4e.gif', 'pythonPowered.gif',
                      'python_conf_ora.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 запущен как
                                                    # самостоятельный сценарий
```

Если запустить этот сценарий, он сначала создаст интерфейс, изображенный на рис. 9.10. На рис. 9.11 изображено это же окно после того как оно было несколько растянуто, с оторванным меню Image и выбранным меню Edit. Панель инструментов в нижней части окна растягивается вместе с окном только по горизонтали. Обратите также внимание, что этот сценарий изменяет форму указателя мыши, когда он находится над панелью инструментов. Поэкспериментируйте с ним самостоятельно, чтобы получить более полное представление о том, как он действует.

Использование изображений в панелях инструментов

Как видно на рис. 9.11, пункты меню легко могут быть украшены графическими изображениями. Хотя это и не было продемонстрировано в примере 9.8, тем не менее элементы панелей инструментов могут также снабжаться картинками, как и элементы меню Image в примере. Для этого достаточно просто поместить небольшие изображения на кнопки в панели инструментов, как мы делали это в примерах с миниатюрами, в последнем разделе главы 8. Как вы уже знаете, при наличии предварительно созданных изображений для кнопок на панели инструментов не составляет никакого труда ассоциировать их с кнопками. Фактически для динамического создания таких изображений требуется приложить ненамного больше труда – навыки создания миниатюр с помощью пакета PIL, полученные нами в предыдущей главе, придутся кстати и в этом контексте.

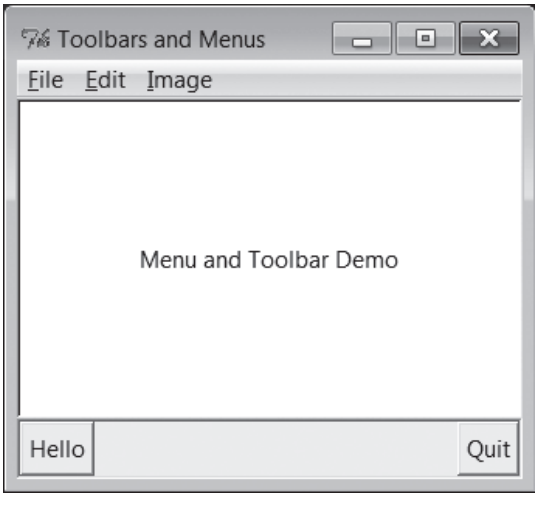


Рис. 9.10. *Сценарий menuDemo: меню и панели инструментов*

Рис. 9.11. *Изображения в меню и отрывные меню в действии*

Для иллюстрации сказанного убедитесь, что у вас расширение PIL установлено, и замените метод конструирования панели инструментов в примере 9.8 следующим фрагментом (я выполнил такую замену

в файле *menuDemo2.py* в пакете с примерами, поэтому вы можете запускать его и экспериментировать с ним):

```
# изменяет размеры изображений для кнопок на панели инструментов с помощью PIL

def makeToolBar(self, size=(40, 40)):
    from PIL.ImageTk import PhotoImage, Image    # для jpeg или новых миниатюр
    imgdir = r'../PIL/images/'
    toolbar = Frame(self, cursor='hand2', relief=SUNKEN, bd=2)
    toolbar.pack(side=BOTTOM, fill=X)
    photos = 'ora-lp4e-big.jpg', 'PythonPoweredAnim.gif',
             'python_conf_ora.gif'
    self.toolPhotoObjs = []
    for file in photos:
        imgobj = Image.open(imgdir + file)       # создать новую миниатюру
        imgobj.thumbnail(size, Image.ANTIALIAS)  # фильтр с лучшим качеством
        img = PhotoImage(imgobj)
        btn = Button(toolbar, image=img, command=self.greeting)
        btn.config(relief=RAISED, bd=2)
        btn.config(width=size[0], height=size[1])
        btn.pack(side=LEFT)
        self.toolPhotoObjs.append((img, imgobj)) # сохранить ссылку
    Button(toolbar, text='Quit', command=self.quit).pack(side=RIGHT, fill=Y)
```

Если запустить этот альтернативный сценарий, он создаст окно, изображенное на рис. 9.12, – три пункта меню Image, присутствующего в верхней части окна, теперь также доступны в виде кнопок на панели инстру-

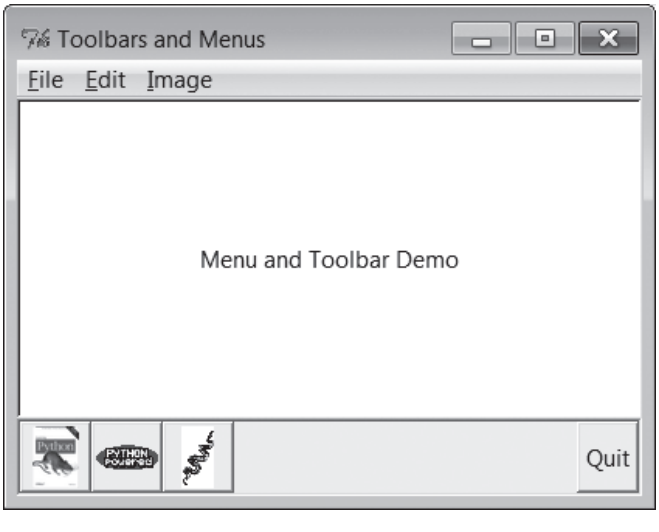


Рис. 9.12. *Сценарий menuDemo2: добавление изображений в панель инструментов с помощью PIL*

ментов в нижней части окна, наряду с кнопкой завершения программы, содержащей текстовую метку. Как и прежде, указатель мыши меняет свою форму при наведении на кнопки в панели инструментов.

Пакет PIL можно не использовать при наличии изображений в формате GIF или в поддерживаемом растровом формате, созданных вручную. Достаточно просто загружать изображения из файлов, используя стандартный объект `PhotoImage` из библиотеки tkinter, как показано в следующей альтернативной реализации метода конструирования панели инструментов (эта версия сценария сохранена в файле *menuDemo3.py* в пакете с примерами):

```
# использует подготовленные изображения gif и стандартные средства tkinter

def makeToolBar(self, size=(30, 30)):
    imgdir = r'../gifs/'
    toolbar = Frame(self, cursor='hand2', relief=SUNKEN, bd=2)
    toolbar.pack(side=BOTTOM, fill=X)
    photos = 'ora-lp4e.gif', 'pythonPowered.gif', 'python_conf_ora.gif'
    self.toolPhotoObjs = []
    for file in photos:
        img = PhotoImage(file=imgdir + file)
        btn = Button(toolbar, image=img, command=self.greeting)
        btn.config(bd=5, relief=RIDGE)
        btn.config(width=size[0], height=size[1])
        btn.pack(side=LEFT)
        self.toolPhotoObjs.append(img)          # сохранить ссылку
Button(toolbar, text='Quit', command=self.quit).pack(side=RIGHT, fill=Y)
```

Если запустить эту альтернативную версию, использующую изображения в формате GIF, она воспроизведет окно, изображенное на рис. 9.13. В зависимости от предпочтений ваших пользователей вам может потребоваться изменить размеры изображений GIF, используемых в этом качестве, с помощью других инструментов – изображения на кнопках фиксированного размера здесь выводится только частично, что может оказаться нежелательным.

Данный пример является первым приблизительным решением для создания кнопок с картинками на панели инструментов. Существует множество других способов настройки таких кнопок. Однако, поскольку мы собираемся еще вернуться к пакету PIL далее в этой главе, когда будем исследовать холсты, мы оставим дальнейшую доработку сценария для самостоятельного упражнения.

Автоматизация создания меню

Меню служат мощным инструментом создания интерфейса в tkinter. Однако если вы рассуждаете, как я, примеры из этого раздела могут показаться вам слишком трудоемкими. Конструирование меню требует писать много программного кода, и велика вероятность ошибок. Лучше

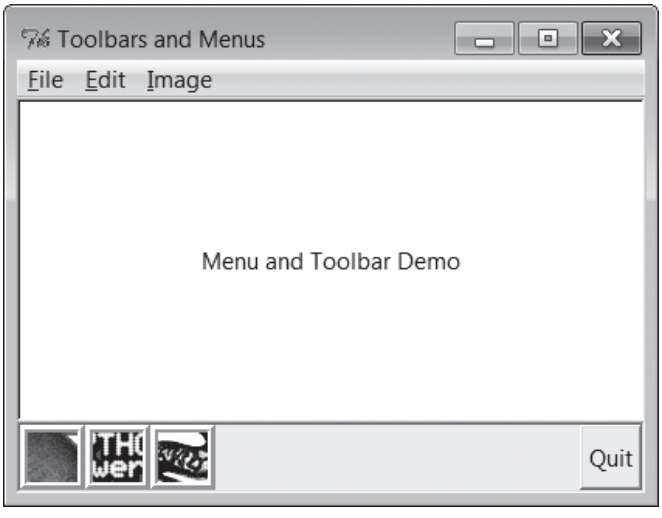


Рис. 9.13. *Сценарий menuDemo3: заранее подготовленные изображения GIF в панели инструментов*

было бы автоматически строить и связывать меню, описывая их содержимое на более высоком уровне. В главе 10 мы познакомимся с инструментом, носящим название GuiMixin, который автоматизирует процесс создания меню исходя из структуры данных, описывающей все необходимые меню. Дополнительным преимуществом этого инструмента является поддержка обоих типов меню – оконных и на основе фреймов, благодаря чему они могут использоваться как в самостоятельных программах, так и во вложенных компонентах. Важно понимать, какие вызовы лежат в основе создания меню, но не обязательно запоминать их навсегда.

Виджеты Listbox и Scrollbar

Возобновим наш обзор виджетов. Виджеты `Listbox` позволяют отображать списки элементов, доступных для выбора, а виджеты `Scrollbar` предназначаются для прокрутки содержимого других виджетов. Эти виджеты часто используются друг с другом, поэтому будем изучать их одновременно. В примере 9.9 виджеты `Listbox` и `Scrollbar` образуют упакованный набор.

Пример 9.9. PP4E\Gui\Tour\scrolledlist.py

```
"простой настраиваемый компонент окна списка с прокруткой"

from tkinter import *

class ScrolledList(Frame):
```

```
    def __init__(self, options, parent=None):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)     # сделать растягиваемым
        self.makeWidgets(options)

    def handleList(self, event):
        index = self.listbox.curselection()  # при двойном щелчке на списке
        label = self.listbox.get(index)      # извлечь выбранный текст
        self.runCommand(label)               # и вызвать действие
                                             # или get(ACTIVE)
    def makeWidgets(self, options):
        sbar = Scrollbar(self)
        list = Listbox(self, relief=SUNKEN)
        sbar.config(command=list.yview)      # связать sbar и list
        list.config(yscrollcommand=sbar.set) # сдвиг одного = сдвиг другого
        sbar.pack(side=RIGHT, fill=Y)        # первым добавлен – посл. обрезан
        list.pack(side=LEFT, expand=YES, fill=BOTH) # список обрезается первым
        pos = 0
        for label in options:                # добавить в виджет списка
            list.insert(pos, label)          # или insert(END,label)
            pos += 1                         # или enumerate(options)
       #list.config(selectmode=SINGLE, setgrid=1) # режимы выбора, измен. разм.
        list.bind('<Double-1>', self.handleList)  # установить обр-к события
        self.listbox = list

    def runCommand(self, selection):         # необходимо переопределить
        print('You selected:', selection)

if __name__ == '__main__':
    options = (('Lumberjack-%s' % x) for x in range(20)) # или map/lambda,
    ScrolledList(options).mainloop()                     # [...]
```

Этот модуль можно запускать как самостоятельный сценарий, чтобы поэкспериментировать с этими виджетами, или использовать в качестве библиотечного объекта. Передавая различные списки выбора в аргументе `options` и переопределяя метод `runCommand` в подклассе, можно повторно использовать определенный здесь класс компонента `ScrolledList` всякий раз когда потребуется вывести список с прокруткой. Мы еще будем использовать этот класс в главе 11, в примере программы PyEdit. При грамотном подходе можно легко расширить библиотеку tkinter классами на языке Python таким способом.

Если запустить этот пример как самостоятельный сценарий, он создаст окно, подобное изображенному на рис. 9.14, которое было получено в Windows 7. Это фрейм `Frame` со списком `Listbox` в левой части, содержащим 20 сгенерированных элементов (на пятом выполнен щелчок) и связанным с виджетом `Scrollbar` в правой части, предназначенным для прокрутки списка. Если переместить ползунок в полосе прокрутки, список также будет прокручиваться, и наоборот.

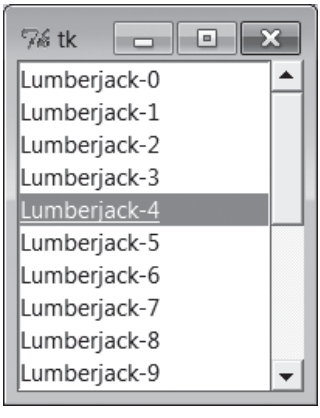


Рис. 9.14. *Сценарий scrolledlist в действии*

Программирование виджетов списков

Виджеты списков достаточно просты в использовании, но в сравнении с виджетами, которые рассматривались до сих пор, они заполняются и обрабатываются довольно своеобразными способами. Многие методы виджета списка принимают индекс, указывающий на элемент списка. Нумерация элементов начинается с 0, но библиотека tkinter вместо целочисленных смещений принимает также особые строки имен: `end` – для ссылки на конец списка, `active` – для обозначения выбранной строки и другие. Поэтому обращение к виджету списка обычно можно оформить несколькими способами.

Например, следующий сценарий добавляет элементы к списку, находящемуся в окне, вызывая его метод `insert` и последовательно увеличивая смещение (начиная с нуля – эту операцию можно было бы автоматизировать с помощью встроенной функции `enumerate`):

```
list.insert(pos, label)
pos = pos + 1
```

Но список можно также заполнить, добавляя элементы в конец, вообще не используя счетчик, – с помощью любой из следующих инструкций:

```
list.insert('end', label) # добавление в конец: подсчет позиций не нужен
list.insert(END, label)   # END - константа со значением 'end' в tkinter
```

Виджет `Listbox` не имеет параметра, такого как `command`, с помощью которого обычно регистрируются обработчики событий нажатий кнопок, поэтому выделенные в нем элементы следует получать во время обработки событий других виджетов (например, щелчков на кнопках) либо обрабатывать сделанные пользователем выделения, внедряясь в протоколы других событий. Чтобы получить выбранное значение, в следующем сценарии выполняется привязка обработчика события `<Double-1>`

двойного щелчка левой кнопкой мыши с помощью bind (рассмотренного ранее в этом обзоре).

В обработчике двойного щелчка этот сценарий получает выделенный в списке элемент с помощью следующей пары методов:

```
index = self.listbox.curselection() # получить индекс выделенного элемента
label = self.listbox.get(index)     # текст, соответствующий этому индексу
```

Эту операцию можно реализовать иначе. 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 на элементе списка – сценарий получает его в виде строки с текстом выбранного элемента:

```
C:\...\PP4E\Gui\Tour> python scrolledlist.py
You selected: Lumberjack-2
You selected: Lumberjack-19
You selected: Lumberjack-4
You selected: Lumberjack-12
```

Виджеты списков могут служить отличными инструментами ввода данных даже без полос прокрутки. Они принимают также параметры настройки, определяющие цвет, шрифт и рельеф. Наряду с режимом выбора единственного элемента они также поддерживают возможность выбора нескольких элементов одновременно. По умолчанию используется режим выбора единственного элемента, но вы можете передать в аргументе selectmode четыре значения: SINGLE, BROWSE, MULTIPLE и EXTENDED (по умолчанию: BROWSE). Первые два из них определяют режимы выбора единственного элемента, а последние два позволяют выбирать сразу несколько элементов.

Эти режимы имеют очень тонкие отличия. Например, режим BROWSE напоминает SINGLE, но дополнительно позволяет перетаскивать выделенный элемент. Щелчок на элементе в режиме MULTIPLE изменяет его состояние, не оказывая влияния на состояние других элементов. Режим EXTENDED также позволяет выбирать несколько элементов, но использует порядок выделения, как принято в интерфейсе проводника файлов Windows – первый элемент выбирается простым щелчком, несколько элементов – щелчком, при удерживаемой клавише Ctrl, а диапазон элементов – щелчком, при удерживаемой клавише Shift. Режим множественного выбора можно реализовать, как показано ниже:

```
listbox = Listbox(window, bg='white', font=('courier', fontsz))
listbox.config(selectmode=EXTENDED)
listbox.bind('<Double-1>', (lambda event: onDoubleClick()))
```

```
# onDoubeClick: извлекает сообщения, выбранные в списке
selections = listbox.curselection()         # кортеж строк чисел, 0..N-1
selections = [int(x)+1 for x in selections] # преобразует в int,
                                            # переводит в диапазон 1..N
```

В режиме множественного выбора метод `curselection` возвращает список строк цифр, соответствующих позициям выбранных элементов, – если не выбран ни один элемент, возвращается пустой кортеж. В действительности этот метод всегда возвращает кортеж строк цифр, даже в режиме выбора единственного элемента (нам не пришлось беспокоиться об этом в примере 9.9, потому что метод `get` виджета при извлечении значения корректно воспринимает кортеж с одним элементом).

Вы можете самостоятельно поэкспериментировать с альтернативными режимами выбора, раскомментировав строку в примере 9.9, где определяется параметр `selectmode`, и изменяя значение. При этом двойной щелчок мышью в режиме множественного выбора может порождать сообщение об ошибке, потому что методу `get` будет передаваться кортеж более чем с одним индексом выбранного элемента (выведите его, чтобы убедиться в этом). Режим множественного выбора мы будем использовать в примере PyMailGUI, далее в этой книге (в главе 14), поэтому дальнейшее обсуждение этой темы я отложу до будущих примеров.

Программирование полос прокрутки

Однако самое большое таинство в примере 9.9 свершается в следующих двух строках:

```
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 производится связывание полосы прокрутки и окна списка – их значения просто ссылаются на связанные методы друг друга. Благодаря такому соединению библиотека tkinter автоматически синхронизирует два виджета при перемещениях в них. Вот как это действует:

- Перемещение полосы прокрутки вызывает обработчик, зарегистрированный с помощью ее параметра `command`. Здесь `list.yview` ссылается на встроенный метод виджета списка, который пропорционально настраивает отображение списка, исходя из аргументов, переданных обработчику.
- При вертикальном перемещении в окне списка вызывается обработчик, зарегистрированный в его параметре `yscrollcommand`. В данном сценарии встроенный метод `sbar.set` пропорционально настраивает полосу прокрутки.

Иными словами, прокрутка в одном виджете автоматически вызывает прокрутку 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, а также параметры

yscrollcommand и xscrollcommand, в которых определяются обработчики связанной с ними полосы прокрутки. У полос прокрутки есть параметр command, в котором указывается обработчик, вызываемый при прокрутке. Библиотека tkinter передает этим методам информацию, определяющую новое положение (например, «прокрутить вниз на 10%»), но программисту не требуется опускаться в сценариях до таких деталей.

Так как полоса прокрутки и окно списка взаимно связаны путем установки их параметров, при перемещении движка полосы прокрутки автоматически происходит прокрутка содержимого списка, а при прокрутке содержимого списка автоматически перемещается движок в полосе прокрутки. Чтобы выполнить перемещение с помощью полосы прокрутки, нужно перетащить ее движок либо щелкнуть на стрелке или в пустой области. Чтобы выполнить прокрутку в списке, следует щелкнуть на нем и использовать клавиши со стрелками или переместить указатель мыши выше или ниже окна, не отпуская кнопки мыши. В обоих случаях список и полоса прокрутки двигаются в унисон. На рис. 9.15 показано, что произойдет после перемещения в списке на несколько элементов вниз тем или иным способом.

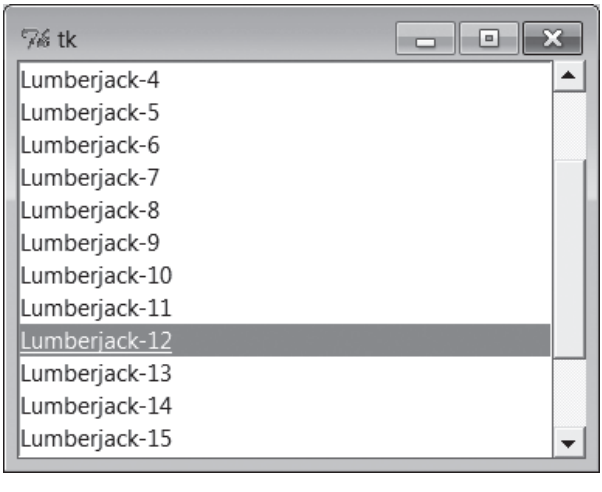


Рис. 9.15. *Прокрутка до середины списка*

Компоновка полос прокрутки

Наконец, вспомним, что виджеты, присоединяемые к интерфейсу последними, всегда обрезаются первыми при уменьшении размеров окна. По этой причине важно добавлять полосы прокрутки как можно раньше, чтобы они исчезли последними, когда окно уменьшится до таких размеров, что в нем ничего нельзя будет показать. Обычно можно справиться с тем, что окно списка выведено не полностью, но полоса прокрутки необходима для перемещения по списку. Как показано на

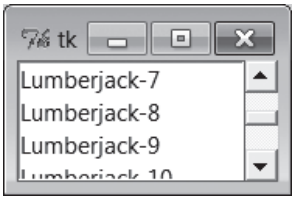


Рис. 9.16. *Список уменьшился*

рис. 9.16, при уменьшении окна сценария отрезается часть списка, но полоса прокрутки сохраняется.

В то же время, обычно не требуется, чтобы полоса прокрутки расширялась вместе с окном, поэтому компоновка ее должна выполняться с одним параметром `fill=Y` (или `fill=X` для прокрутки по горизонтали), без `expand=YES`. В частности, расширение окна этого примера увеличивает окно списка, но изменяет ширину полосы прокрутки, прикрепленной справа.

В примерах этой и последующих глав мы неоднократно будем встречаться с полосами прокрутки и виджетами списков (можно заглянуть вперед и посмотреть примеры PyEdit, PyMailGUI, PyForm, PyTree и ShellGui). И хотя основы их применения охватываются в данной главе, следует отметить, что за кадром осталось многое, что могут предложить эти виджеты.

Например, столь же легко к прокручиваемым виджетам можно добавить горизонтальные полосы прокрутки. Они программируются почти так же, как вертикальные, только имена обработчиков начинаются с «x», а не «y» (например, `xscrollcommand`), а для объекта полосы прокрутки устанавливается параметр `orient='horizontal'`. Чтобы добавить сразу две полосы прокрутки, вертикальную и горизонтальную, и связать их с виджетом, можно использовать такой прием:

```
window  = Frame(self)
vscroll = Scrollbar(window)
hscroll = Scrollbar(window, orient='horizontal')
listbox = Listbox(window)

# прокрутить список при перемещении движка в полосе прокрутки
vscroll.config(command=listbox.yview, relief=SUNKEN)
hscroll.config(command=listbox.xview, relief=SUNKEN)

# переместить движок в полосе прокрутки при прокрутке списка
listbox.config(yscrollcommand=vscroll.set, relief=SUNKEN)
listbox.config(xscrollcommand=hscroll.set)
```

Смотрите пример использования холста для вывода изображений далее в этой главе, а также в программах PyEdit, PyTree и PyMailGUI далее в этой книге, где демонстрируется использование горизонтальных по-

лос прокрутки. Полосы прокрутки могут действовать в графических интерфейсах различными способами – их можно связывать с виджетами других типов. Например, их часто прикрепляют к виджету Text, что приводит нас к следующей теме данного обзора.

Виджет Text

Уже отмечалось, что наиболее сильными сторонами библиотеки tkinter являются виджеты Text и Canvas. Оба они обладают богатым набором функций. Например, виджет Text оказался достаточно мощным, чтобы на его основе можно было создать веб-броузер Grail – экспериментальный веб-броузер, реализованный на языке Python. Виджет Text поддерживает сложные настройки стилей шрифтов, позволяет встраивать графику, имеет неограниченное количество уровней отката и возврата изменений и многое другое. Виджет Canvas, универсальный графический инструмент, позволяет отображать произвольные изображения и также положен в основу многих приложений для сложной обработки изображений и визуализации.

В главе 11 мы воспользуемся двумя этими виджетами для реализации текстового редактора (PyEdit), графического редактора (PyDraw), часов с графическим интерфейсом (PyClock) и программ для просмотра изображений (PyPhoto и PyView). Однако в этой, экскурсионной главе мы будем использовать эти виджеты в более простых примерах. В примере 9.10 реализован простой интерфейс отображения текста с прокруткой, который может вывести строку текста или файл.

Пример 9.10. PP4E\Gui\Tour\scrolledtext.py

```
"простой компонент просмотра текста или содержимого файла"

print('PP4E 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 растягиваемым
        self.makewidgets()
        self.settext(text, file)

    def makewidgets(self):
        sbar = Scrollbar(self)
        text = Text(self, relief=SUNKEN)
        sbar.config(command=text.yview)        # связать sbar и text
        text.config(yscrollcommand=sbar.set)   # сдвиг одного = сдвиг другого
        sbar.pack(side=RIGHT, fill=Y)        # первым добавлен - посл. обрезан
        text.pack(side=LEFT, expand=YES, fill=BOTH) # Text обрезается первым
        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 стр. 1, кол. 0
        self.text.mark_set(INSERT, '1.0')      # установить курсор вставки
        self.text.focus()                      # сэкономить щелчок мышью

    def gettext(self):                         # возвращает строку
        return self.text.get('1.0', END+'-1c') # от начала до конца

if __name__ == '__main__':
    root = Tk()
    if len(sys.argv) > 1:
        st = ScrolledText(file=sys.argv[1])    # имя файла в командной строке
    else:
        st = ScrolledText(text='Words\ngo here') # иначе: две строки
    def show(event):
        print(repr(st.gettext()))              # вывести как простую строку
    root.bind('<Key-Escape>', show)            # esc = выводит дамп текста
    root.mainloop()
```

Подобно объекту `ScrolledList` из примера 9.9 объект `ScrolledText` в этом файле создавался как многократно используемый компонент, но точно так же этот сценарий может выполняться автономно, выводя содержимое текстового файла. Так же, как в предыдущем разделе, этот сценарий сначала прикрепляет полосу прокрутки, чтобы при сжатии окна она исчезала последней, и настраивает встроенный объект `Text`, чтобы он растягивался в обоих направлениях при увеличении размеров окна. Если при запуске передать сценарию аргумент с именем файла, он создаст окно, изображенное на рис. 9.17: сценарий встраивает виджет `Text` в левую часть окна, а связанную с ним полосу прокрутки – в правую.

Для забавы я заполнил текстовый файл[1], отображаемый в окне, с помощью следующего сценария и команд (и не только потому, что я жил в Колорадо рядом с отелем, пользующимся дурной славой):

```
C:\...\PP4E\Gui\Tour> type makefile.py
f = open('jack.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:\...\PP4E\Gui\Tour> python makefile.py

C:\...\PP4E\Gui\Tour> python scrolledtext.py jack.txt
PP4E scrolledtext
```

[1] Использована цитата из фильма Стенли Кубрика «Сияние». – *Прим. ред.*

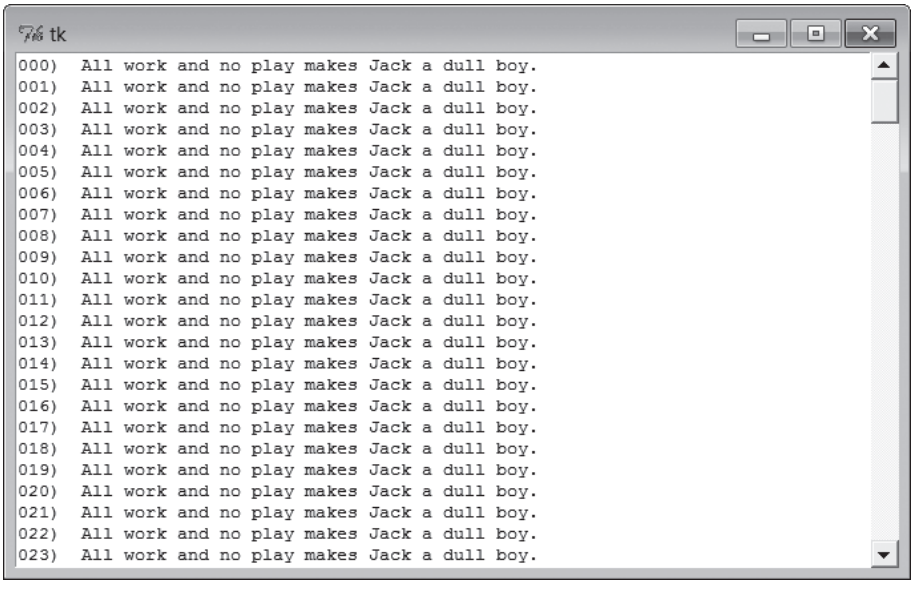


Рис. 9.17. *Сценарий scrolledtext в действии*

Для просмотра файла его имя нужно передать сценарию в аргументе командной строки – текст файла будет автоматически выведен в новом окне. По умолчанию он выводится шрифтом, который может оказаться разным на разных платформах (и может оказаться не моноширинным в некоторых их них), но в следующем примере мы устраним это несоответствие, передав виджету `Text` параметр `font`. Нажатие клавиши Escape вызывает извлечение и вывод всего текстового содержимого виджета в виде единственной строки (подробнее об этом чуть ниже).

Обратите внимание на сообщение `PP4E scrolledtext`, которое выводится при выполнении сценария. Поскольку в стандартном дистрибутиве Python также есть файл *scrolledtext.py* (в модуле `tkinter.scrolledtext`) с совершенно иной реализацией и интерфейсом, данный сценарий идентифицирует себя при выполнении или импортировании, чтобы можно было отличить один от другого. Если сценарий стандартной библиотеки когда-либо будет исключен из дистрибутива, импортируйте тот, что приведен здесь, чтобы получить простое средство просмотра текста, и измените вызовы настройки параметров, чтобы они содержали спецификатор `.text` (например, `x.text.config` вместо `x.config` – библиотечная версия создает подкласс `Text`, а не `Frame`).

Программирование виджета Text

Чтобы понять, как вообще работает этот сценарий, необходимо разобраться с некоторыми особенностями виджета `Text`. Мы уже знакомы

с виджетами Entry и Message, которые охватывают некоторое подмножество возможных применений виджета Text. В отличие от них, виджет Text обладает более широкими функциональными возможностями – он поддерживает ввод и отображение нескольких строк текста, операции редактирования для программ и интерактивных пользователей, различные шрифты и цвета и многое другое. Объекты Text создаются, настраиваются и прикрепляются к графическому интерфейсу так же, как любые другие виджеты, но у них есть особые свойства.

Текст является строкой Python

Несмотря на всю мощь виджета Text, его интерфейс можно свести к двум базовым понятиям. Во-первых, содержимое элемента Text представляется в сценариях Python в виде единой строки, где несколько строчек разделяются обычным символом \n завершения строки. Например, строка 'Words\ngo here' представляет две строчки при записи в виджет Text и при получении из него. Обычно текстовое содержимое имеет замыкающий символ \n, но это не обязательно.

Чтобы проиллюстрировать эту особенность, данный сценарий связывает событие нажатия клавиши Escape с обработчиком, который получает и выводит все содержимое виджета Text:

```
C:\...\PP4E\Gui\Tour> python scrolledtext.py
PP4E scrolledtext
'Words\ngo here'
'Always look\non the bright\nside of life\n'
```

Если запустить этот сценарий с аргументом, он запишет в виджет Text все содержимое файла. При запуске без аргументов сценарий вставит в виджет простую литеральную строку, отображаемую при первом нажатии Escape (напомню, что \n является экранированной последовательностью, соответствующей символу конца строки). Второй вывод был получен после редактирования текста, нажатием клавиши Escape в уменьшенном окне, изображенном на рис. 9.18.

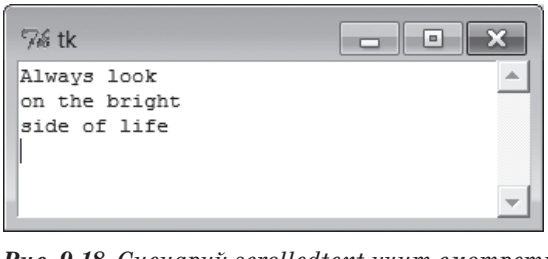


***Рис. 9.18.** Сценарий scrolledtext учит смотреть на жизнь позитивно*

Позиция в строке

Вторым ключом к пониманию виджета Text является определение позиции в строке текста. Как и виджеты списков, виджеты Text позволя-

ют указывать эту позицию различными способами. Методы Text, ожидающие получить позицию, принимают индекс, метку или тег. Кроме того, некоторые особые операции вызываются с предопределенными метками или тегами – текстовый курсор вставки имеет метку INSERT, а текущее выделение имеет тег SEL. Так как они являются фундаментальными при работе с виджетом Text и обеспечивают большую часть его выразительности, рассмотрим поближе эти параметры настройки.

Индексы в Text. Будучи многострочным виджетом, Text поддерживает индексы строк и колонок. Например, рассмотрим интерфейсы базовых операций ввода, удаления и извлечения 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 от начала до конца
```

Во всех этих предложениях первый аргумент является абсолютным индексом, ссылающимся на начало строки с текстом: строка '1.0' означает строку 1, колонку 0 (строки нумеруются с 1, а колонки – с 0, однако аргумент '0.0' также считается допустимым и интерпретируется как ссылка на начало текста). Индекс '2.1' ссылается на второй символ во второй строке.

Как и в случае с виджетом списка, при работе с текстовым виджетом также можно использовать символические имена: END в предшествующем вызове метода delete указывает на позицию, находящуюся сразу за последним символом в строке текста (библиотека tkinter записывает в эту переменную 'end'). Аналогично символический индекс INSERT (на самом деле строка 'insert') ссылается на позицию, находящуюся сразу за курсором вставки, – место, где будут появляться символы, вводимые с клавиатуры. Символические имена, такие как INSERT, можно также назвать метками, которые будут описаны ниже.

Дополнительной точности можно достичь, добавляя к строкам индексов простые арифметические расширения. Индексное выражение END+'-1c' в вызове метода get в предыдущем примере в действительности является строкой 'end-1c' и ссылается на позицию за один символ до 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 символ \n, который виджет добавляет к своему содержимому (и который может добавить пустую строку при сохранении в файле).

Аналогичные расширения строк индексов позволяют ссылаться на символы, находящиеся впереди (+1c), строки, находящиеся впереди и позади (+2l, -2l), а также на концы строк и начала слов, в которых находится индекс (lineend, wordstart). Индексы передаются большинству методов виджета Text.

Метки в Text. Помимо идентификаторов строка/колонка, позиции можно определять в виде имен меток – символических имен позиций между двумя символами. В отличие от абсолютных позиций строка/колонка, 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 другую строку или колонку.

Для создания метки используется метод `mark_set` объекта `Text`, которому передается строка имени и индекс, определяющий ее логическое положение. Например, этот сценарий устанавливает текстовый курсор вставки 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 предопределенной меткой, идентифицирующей позицию текстового курсора вставки, – изменение ее влечет изменение позиции курсора вставки. Чтобы создать собственную метку, достаточно просто передать уникальное имя, как показано во втором вызове выше, и использовать его везде, где требуется указать позицию в тексте. Вызов метода `mark_unset` удаляет метку по имени.

Теги в Text. Кроме абсолютных индексов и символических имен меток виджет `Text` поддерживает понятие тегов – символических имен, ассоциируемых с одной или несколькими подстроками в текстовом содержимом виджета `Text`. Теги можно применять с разными целями, в том числе для представления позиции там, где это требуется: помеченные тегами элементы определяются индексами начала и конца, которые можно впоследствии передавать методам, требующим указания позиции.

Например, библиотека tkinter предоставляет встроенный тег с именем `SEL` – переменную с предопределенным строковым значением `'sel'`, – которое автоматически ссылается на текст, выделенный в данный момент. Чтобы получить текст, выделенный (подсвеченный) с помощью мыши, вызовите любой из следующих методов:

```
text = self.text.get(SEL_FIRST, SEL_LAST)     # теги для индексов от/до
text = self.text.get('sel.first', 'sel.last') # или строки и константы
```

Имена `SEL_FIRST` и `SEL_LAST` являются обычными переменными в модуле `tkinter` с предопределенными значениями, используемыми во втором вызове. Метод `get` ожидает получить два индекса. Чтобы получить текст по тегу, добавьте к его имени расширения `.first` и `.last`, которые дают индексы начала и конца.

Чтобы пометить тегом подстроку, можно вызвать метод `tag_add` виджета `Text`, передав ему строку с именем тега и позиции начала и конца (тегами можно помечать текст, добавляемый методом `insert`). Чтобы снять

тег со всех символов в некоторой области текста, можно вызвать метод tag_remove:

```
self.text.tag_add('alltext', '1.0', END) # пометить тегом весь текст
self.text.tag_add(SEL, index1, index2)   # выделить от index1 до index2
self.text.tag_remove(SEL, '1.0', END)    # снять выделение со всего текста
```

Здесь в первой строке создается новый тег для всего текста в виджете – от начальной до конечной позиции. Во второй строке во встроенный тег выделения SEL добавляется диапазон символов – эти символы автоматически подсвечиваются, поскольку для этого тега предопределена такая настройка его элементов. В третьей строке все символы текста исключаются из тега SEL (снимаются все выделения). Обратите внимание, что метод tag_remove просто снимает тег с текста в указанном диапазоне – чтобы полностью удалить тег, нужно вызвать метод tag_delete. Кроме того, имейте в виду, что эти методы применяются к самим тегам – чтобы удалить фактический текст, следует использовать метод delete, представленный выше.

Можно также динамически отображать индексы в теги. Например, метод search возвращает индекс row.column первого вхождения строки между начальной и конечной позициями. Чтобы автоматически выделить найденный текст, его индекс следует добавить 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 и выделить найденную строку
self.text.focus()                      # выбрать сам виджет Text
```

Если требуется выделить только одну строку, нужно сначала вызвать метод tag_remove, о котором говорилось выше, – этот фрагмент добавляет новое выделение к тем, которые уже существуют (таким способом на экране можно создать нескольких выделений). Вообще в тег можно добавить любое количество подстрок и обрабатывать их группой.

Подведем итоги: индексы, метки и позиции тегов можно использовать везде, где требуется указать позицию в тексте. Например, метод see прокручивает содержимое, пока требуемая позиция не окажется в области видимости, – он позволяет указывать позиции всеми тремя способами:

```
self.text.see('1.0')     # прокрутить вверх
self.text.see(INSERT)    # прокрутить до метки курсора вставки
self.text.see(SEL_FIRST) # прокрутить до тега выделения
```

Теги также могут применяться для форматирования и привязки событий, но эти подробности будут обсуждаться в конце раздела.

Операции редактирования текста

В примере 9.11 используются некоторые из этих понятий. Он расширяет пример 9.10, добавляя поддержку четырех наиболее часто используемых операций редактирования – сохранение в файл, удаление и встав-

ка текста и поиск строки – за счет создания подкласса, наследующего класс ScrolledText, с дополнительными кнопками и методами. В виджете Text есть набор готовых привязок клавиш, выполняющих некоторые часто используемые операции редактирования, но они могут оказаться совсем не теми, которые можно было бы ожидать на конкретных платформах. Чаще в текстовом редакторе с графическим интерфейсом предоставляются специальные элементы управления, выполняющие операции редактирования, что более дружественно по отношению к пользователю.

Пример 9.11. PP4E\Gui\Tour\simpleedit.py

```
"""
за счет наследования добавляет в ScrolledText типичные инструменты
редактирования; аналогичного результата можно было бы добиться, применив прием
композиции (встраивания); ненадежно! -- надмножество функций имеется в PyEdit;
"""

from tkinter import *
from tkinter.simpledialog import askstring
from tkinter.filedialog import asksaveasfilename
from quitter import Quitter
from scrolledtext import ScrolledText          # наш, не из библиотеки Python

class SimpleEditor(ScrolledText):              # доп. ф-ции смотрите 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()               # от начала до конца
            open(filename, 'w').write(alltext)     # сохранить текст в файл

    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):                             # добавляет 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) # от позиции курсора
            if where:                                     # вернуть индекс
                print(where)
                pastit = where + ('+%dc' % len(target)) # индекс за целью
               #self.text.tag_remove(SEL, '1.0', END)   # снять выделение
                self.text.tag_add(SEL, where, pastit)   # выделить найденное
                self.text.mark_set(INSERT, pastit)  # установить метку вставки
                self.text.see(INSERT)               # прокрутить текст
                self.text.focus()                   # выбрать виджет Text

if __name__ == '__main__':
    if len(sys.argv) > 1:
        SimpleEditor(file=sys.argv[1]).mainloop()   # имя файла в ком. строке
    else:
        SimpleEditor().mainloop()                   # или нет: пустой виджет
```

Этот сценарий также был написан с оглядкой на многократное использование – определяемый в нем класс `SimpleEditor` можно прикрепить или унаследовать в другой реализации графического интерфейса. Однако, как будет разъяснено в конце раздела, данный пример не настолько надежен, как требуется от библиотечного инструмента общего назначения. Тем не менее в нем реализован действующий текстовый редактор, программный код которого переносим и имеет небольшой объем. Если запустить пример как самостоятельный сценарий, он выведет окно, изображенное на рис. 9.19 (здесь он был запущен в Windows). После каждой успешной операции поиска позиции индексов выводятся в `stdout` – в этом примере логика снятия предыдущего выделения закомментирована, поэтому вторая операция поиска, как видно на рисунке, выделила вторую строку «def», не сняв предыдущее выделение (раскомментируйте эту строку в сценарии, чтобы операция поиска снимала предыдущее выделение):

```
C:\...\PP4E\Gui\Tour> python simpleedit.py simpleedit.py
PP4E scrolledtext
14.4
25.4
```

Операция сохранения выводит имеющийся в библиотеке tkinter стандартный диалог сохранения, который выглядит естественным в каждой из платформ. На рис. 9.20 изображен этот диалог в Windows 7. Операция поиска тоже выводит стандартный диалог для ввода строки поиска (рис. 9.21); в полноценном редакторе можно было бы сохранить эту строку для повторного поиска (что мы и сделаем в главе 11, в реа-

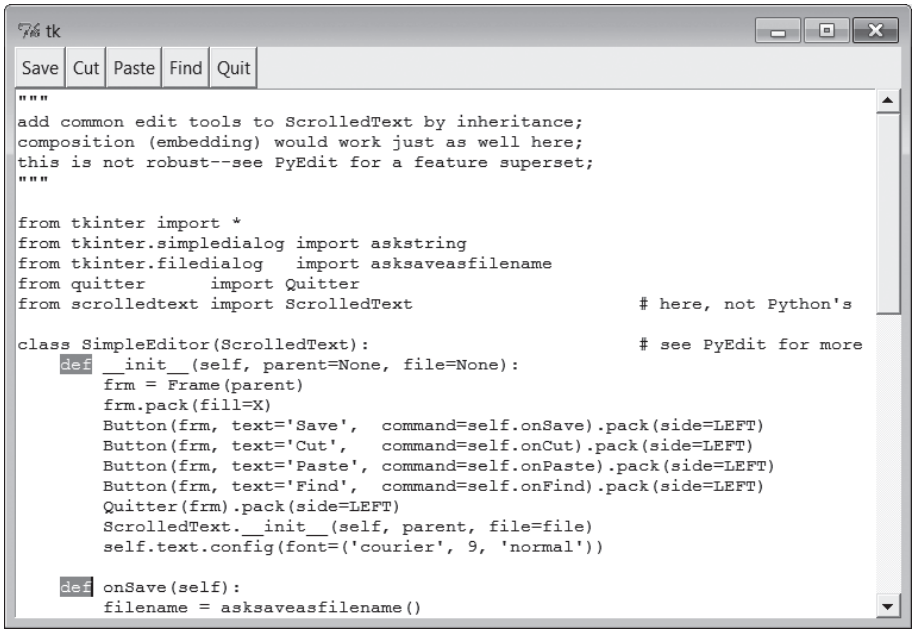


Рис. 9.19. *Сценарий simpleedit в действии*

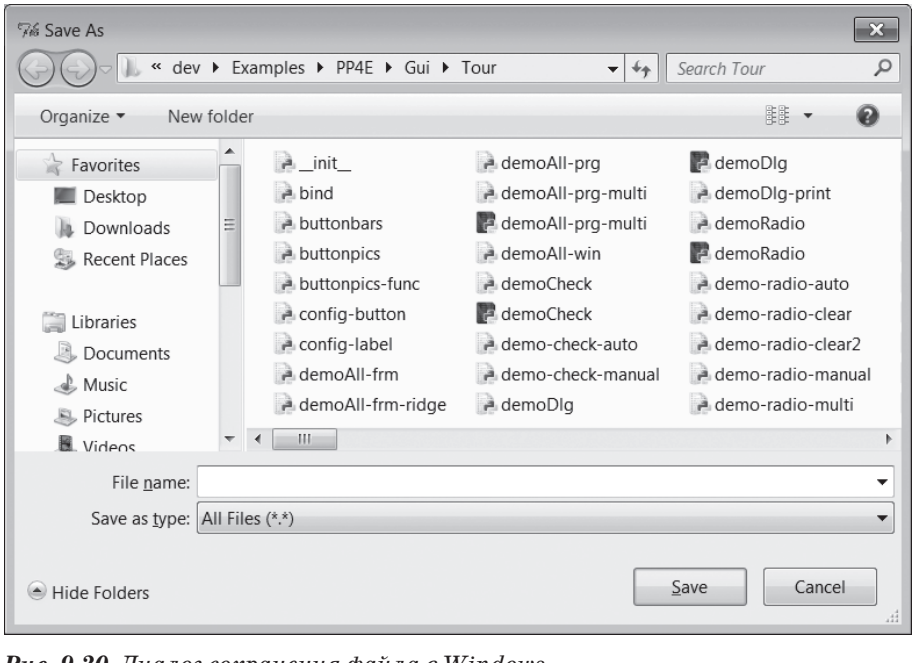


Рис. 9.20. *Диалог сохранения файла в Windows*

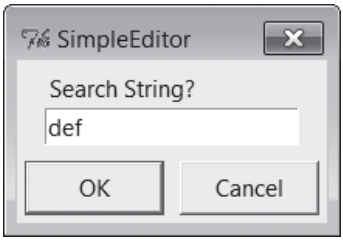


Рис. 9.21. *Диалог поиска*

лизации редактора PyEdit). Для реализации операции завершения повторно был использован компонент кнопки Quit, реализованный нами в главе 8, – этот код гарантирует, что приложение не может быть завершено без подтверждения.

Использование буфера обмена

Помимо операций с виджетом Text в примере 9.11 применяются функции из библиотеки tkinter доступа к буферу обмена для реализации операций удаления и вставки текста. В совокупности эти операции позволяют перемещать текст в файле (вырезать в одном месте и вставлять в другом). Используемый ими буфер обмена служит местом для временного хранения данных – удаленный текст помещается в буфер при вырезании и извлекается оттуда при вставке. Если все происходит в одной программе, нет смысла вырезать строку текста вместо того чтобы просто сохранить ее в обычной переменной Python. Но буфер обмена является значительно более широким понятием.

Буфер обмена, используемый в этом сценарии, является общесистемным хранилищем, совместно используемым всеми программами, выполняющимися в компьютере. Поэтому его можно использовать для передачи данных между приложениями, в числе которых могут быть такие, которые понятия не имеют о библиотеке tkinter. Например, текст, вырезанный или скопированный в Microsoft Word, можно вставлять в окно SimpleEditor, а текст, вырезанный в SimpleEditor, можно вставить в Блокнот (можете попробовать). Используя буфер обмена для реализации операций вырезания и вставки, SimpleEditor автоматически интегрируется с оконной системой в целом. Кроме того, буфер обмена используется не одним только виджетом Text – его можно применять для вырезания и вставки графических объектов в виджете Canvas (обсуждается далее).

Базовый интерфейс к буферу обмена в библиотеке tkinter, использованный в сценарии из примера 9.11, выглядит так:

```
self.clipboard_clear()                          # очистить буфер
self.clipboard_append(text)                     # сохранить строку текста
text = self.selection_get(selection='CLIPBOARD') # получить содержимое, если есть
```

Все эти вызовы доступны в виде методов, наследуемых всеми объектами виджетов tkinter, потому что они разрабатывались как глобальные. Использованное в этом сценарии выделение CLIPBOARD может применяться на всех платформах (существует еще выделение PRIMARY, но обычно им можно пользоваться только в X Window, поэтому мы здесь его не рассматриваем). Обратите внимание, что в случае неудачи метод selection_get возбуждает исключение TclError – данный сценарий ее просто игнорирует и прерывает операцию вставки, но в дальнейшем мы реализуем более удачное решение.

Композиция и наследование

В данном примере класс SimpleEditor использует наследование для расширения ScrolledText дополнительными кнопками и методами обработчиков. Как мы уже видели, допускается также прикреплять (встраивать) объекты графического интерфейса, реализованные как компоненты, подобно ScrolledText. Модель, когда компоненты прикрепляются, обычно называется композицией; существует мнение, что она проще для понимания и реже приводит к конфликту имен, чем расширение наследованием.

Чтобы дать представление о различиях между этими двумя подходами, ниже приводится набросок программного кода, в котором объект ScrolledText прикрепляется к объекту SimpleEditor. Измененные строки выделены в нем полужирным шрифтом (полную реализацию приема композиции можно найти в файле *simpleedit2.py*, в пакете с примерами). В основном задача состоит в передаче правильных родительских элементов и добавлении атрибута st каждый раз, когда требуется получить доступ к методам виджета 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)
        ...часть программного кода опущена...
        Quitter(frm).pack(side=LEFT)
        self.st = ScrolledText(self, file=file)    # прикрепить, не подкласс
        self.st.text.config(font=('courier', 9, 'normal'))

    def onSave(self):
        filename = asksaveasfilename()
        if filename:
            alltext = self.st.gettext()            # доступ 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)
```

...часть программного кода опущена...

При таком подходе нет необходимости наследовать класс `Frame` (виджеты можно было прикреплять непосредственно к родителю), а принадлежность к фреймам позволяет весь этот пакет встраивать и настраивать целиком. Если запустить этот сценарий, он создаст схожее окно. Что лучше – композиция или наследование – судить вам. При правильном использовании классов графического интерфейса они будут работать в любом режиме.

«Простым» он назван обоснованно: PyEdit (забегая вперед)

Наконец, прежде чем вы зарегистрируете в системном реестре редактор `SimpleEditor` как средство по умолчанию для просмотра текстовых файлов, я должен заметить, что хотя он и обладает всеми основными функциями, но является в некотором роде урезанной версией (в действительности – прототипом) редактора PyEdit, с которым мы познакомимся в главе 11. Вы можете захотеть уже сейчас посмотреть пример с PyEdit, если вы ищете более полную реализацию обработки текста на основе библиотеки tkinter. В нем мы будем также использовать более сложные операции с текстом, такие как откат (undo) и возврат (redo) ввода, поиск с учетом регистра символов, поиск во внешних файлах и многие другие. Виджет `Text` обладает такой мощью, что трудно продемонстрировать диапазон его возможностей в программном коде меньшего объема, чем тот, который приведен в программе PyEdit.

Я должен также отметить, что реализация `SimpleEditor` не только ограничена в своих возможностях, но и весьма небрежна – многие граничные случаи проходят без проверки и вызывают неперехватываемые исключения, которые не приводят к завершению графического интерфейса, но и не обрабатываются и не выводят сообщений. Даже о перехваченных ошибках пользователю не сообщается (например, при попытке вставки, когда буфер обмена пуст). Обязательно посмотрите PyEdit, как пример более устойчивой и полной реализации операций, введенных в `SimpleEditor`.

Юникод и виджет Text

Я уже говорил выше, что текстовое содержимое в виджете `Text` всегда представлено в виде строки. Однако в Python 3.X существует два типа строк: `str` – для представления текста Юникода, и `bytes` – для представления строк байтов. Кроме того, текст Юникода может сохраняться в файлы в различных кодировках. Оказывается, что оба эти фактора могут оказывать влияние на порядок использования виджета `Text` в Python 3.X.

В двух словах: виджет `Text` и другие виджеты, так или иначе связанные с текстом, такие как `Entry`, поддерживают возможность отображения национальных наборов символов для обоих типов строк, `str` и `bytes`, но,

чтобы обеспечить поддержку самого широкого диапазона символов, при работе с виджетом необходимо использовать декодированный текст Юникода типа str. В этом разделе мы по полочкам разберем механизмы работы с текстом в библиотеке tkinter, чтобы объяснить причину.

Типы строк в виджете Text

Вы могли обратить внимание на это или нет, но во всех предыдущих примерах текстовое содержимое было представлено в виде строк str – либо жестко определенных в тексте сценариев, либо извлекаемых из простых текстовых файлов и сохраняемых в них, что предполагает использование кодировки по умолчанию для текущей платформы. Однако технически виджет Text позволяет вставлять строки обоих типов, и str и bytes:

```
>>> from tkinter import Text
>>> T = Text()
>>> T.insert('1.0', 'spam')    # вставить строку типа str
>>> T.insert('end', b'eggs')   # вставить строку типа bytes
>>> T.pack()                   # теперь виджет отображает строку "spameggs"
>>> T.get('1.0', 'end')        # извлечь содержимое
'spameggs\n'
```

Возможность вставки текста в виде строки типа bytes может пригодиться при просмотре различных видов текста Юникода, особенно когда имя кодировки не известно. Например, текст, полученный из Интернета (скажем, во вложении к электронному письму или полученный по FTP), может быть представлен в любой кодировке – сохранение его в файлах в двоичном режиме и отображение, как строк типа bytes в виджете Text, может показаться обходным решением проблемы использования кодировок в наших сценариях.

Однако, к большому сожалению, виджет Text возвращает свое содержимое только в виде строки str независимо от того, строки какого типа в него вставлялись, str или bytes, – в любом случае мы получаем обратно уже декодированный текст Юникода:

```
>>> T = Text()
>>> T.insert('1.0', 'Textfileline1\n')
>>> T.insert('end', 'Textfileline2\n') # при вставке str, содержимое - str
>>> T.get('1.0', 'end')                # вызывать pack() не обязательно,
'Textfileline1\nTextfileline2\n\n'     # чтобы обратиться к get()

>>> T = Text()
>>> T.insert('1.0', b'Bytesfileline1\r\n') # для bytes содержимое тоже str!
>>> T.insert('end', b'Bytesfileline2\r\n') # а \r отображается, как пробел
>>> T.get('1.0', 'end')
'Bytesfileline1\r\nBytesfileline2\r\n\n'
```

Фактически мы получаем содержимое виджета в виде строки типа str, даже если мы вставляли строки обоих типов, str и bytes, с единствен-

ным символом \n, добавленным в конец, как показано в первом примере в этом разделе. Ниже приводится более полная иллюстрация:

```
>>> T = Text()
>>> T.insert('1.0', 'Textfileline1\n')
>>> T.insert('end', 'Textfileline2\n')     # добавлены две строки str
>>> T.insert('1.0', b'Bytesfileline1\r\n') # \n добавляется для любого типа
>>> T.insert('end', b'Bytesfileline2\r\n') # pack() отображает 4 строки текста
>>> T.get('1.0', 'end')
'Bytesfileline1\r\nTextfileline1\nTextfileline2\nBytesfileline2\r\n\n'
>>>
>>> print(T.get('1.0', 'end'))
Bytesfileline1
Textfileline1
Textfileline2
Bytesfileline2
```

Это упрощает обработку текста содержимого после его извлечения: мы можем интерпретировать его, как строку типа str независимо от того, строки каких типов вставлялись. Однако эта же особенность усложняет обработку текстовых данных с точки зрения Юникода: мы не можем сохранить возвращаемую строку str в двоичный файл, потому что операция записи в двоичном режиме ожидает получить строку типа bytes. Нам необходимо будет либо закодировать строку в тип bytes вручную, либо открыть файл в текстовом режиме и уповать на то, что строка str сможет быть закодирована. В любом случае нам необходимо будет знать имя применяемой кодировки; положиться на кодировку по умолчанию для данной платформы; в крайнем случае, сделать какие-либо предположения и надеяться, что они оправдаются; или запросить кодировку у пользователя.

Иными словами, даже при том, что библиотека tkinter позволяет вставлять текст в неизвестной кодировке, как строку типа bytes, и просматривать его, тот факт, что содержимое возвращается в виде строки str, в общем случае означает необходимость знать, как кодировать текст при сохранении, чтобы удовлетворить интерфейс файлов в Python 3.X. Кроме того, так как строки bytes, вставляемые в виджеты Text, также должны быть декодируемыми согласно ограничениям поддержки Юникода в библиотеке Tk, будет лучше, если мы будем декодировать текст в строки str самостоятельно, чтобы обеспечить более широкую поддержку Юникода. Чтобы убедиться в правоте этих слов, нам необходимо совершить короткий экскурс в страну Юникода.

Текст Юникода в строках

Причина всех этих сложностей заключается, конечно же, в том, что в мире Юникода мы не можем больше думать о «тексте», не задавая вопрос «какого он вида». Вообще текст может быть закодирован с использованием самых разных схем кодирования. В языке Python это обстоятельство неразрывно связано со строками str и может иметь от-

ношение к строкам bytes, если они содержат кодированный текст. Строки str Юникода в языке Python – это просто строки, но вам придется принимать во внимание кодировки при записи строк в файлы и чтении их из файлов, а также при передаче их в библиотеки, накладывающие ограничения на кодирование текста.

Мы не будем рассматривать здесь проблемы кодирования Юникода во всех подробностях (подробное описание вы найдете в книге «Изучаем Python», а краткое упоминание о том, какое значение это имеет для файлов, – в главе 4), но коротко рассмотрим некоторые положения, чтобы увидеть – какое отношение они имеют к виджетам Text. Для начала вам следует запомнить, что проблемы с текстом ASCII не возникают лишь потому, что ASCII является подмножеством большинства схем кодирования Юникода. Однако данные, выходящие за диапазон представления 7-битовых символов ASCII, в разных схемах кодирования могут быть представлены разными байтами.

Например, следующие строки байтов должны декодироваться с использованием кодировки Latin-1 – попытка использовать кодировку по умолчанию или явно указанную кодировку, не соответствующую строке байтов, будет терпеть неудачу:

```
>>> b = b'A\xc4B\xe4C' # эти байты - текст в кодировке latin-1
>>> b
b'A\xc4B\xe4C'

>>> s = b.decode('utf8')
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: invalid dat...
>>> s = b.decode()
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: invalid dat...

>>> s = b.decode('latin1')
>>> s
'AÄBäC'
```

Декодировав байты в строку Юникода, вы сможете «преобразовать» ее обратно в строку байтов с применением различных кодировок. В действительности при этом будет произведено преобразование в альтернативное двоичное представление, которое позднее мы опять сможем декодировать в строку – по существу строка Юникода не относится к «типу Юникод», к нему могут относиться только двоичные данные:

```
>>> s.encode('latin-1')
b'A\xc4B\xe4C'

>>> s.encode('utf-8')
b'A\xc3\x84B\xc3\xa4C'

>>> s.encode('utf-16')
b'\xff\xfeA\x00\xc4\x00B\x00\xe4\x00C\x00'
```

```
>>> s.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode character '\xc4' in position 1: o...
```

Обратите внимание на последнюю операцию в этом примере: кодируемая строка должна быть совместима с используемой схемой кодирования, в противном случае будет возбуждено исключение. В данном случае диапазон ASCII оказался слишком узким для представления символов, полученных в результате декодирования из байтов в кодировке Latin-1. То есть вы можете преобразовать строку в различные (совместимые) двоичные представления, но тем не менее для декодирования этих данных обратно в строку в общем случае вам необходимо знать кодировку, использовавшуюся при кодировании:

```
>>> s.encode('utf-16').decode('utf-16')
'AÄBäC'
>>> s.encode('latin-1').decode('latin-1')
'AÄBäC'

>>> s.encode('latin-1').decode('utf-8')
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: invalid dat...
>>> s.encode('utf-8').decode('latin-1')
UnicodeEncodeError: 'charmap' codec can't encode character '\xc3' in position 2:...
```

Снова обратите внимание на последнюю операцию. Технически кодирование кодовых пунктов (символов) Юникода в байты UTF-8 и обратное их декодирование с применением кодировки Latin-1 не возбуждает ошибку, но попытка вывести результат приводит к исключению: с точки зрения функции вывода он является зашифрованным мусором. Чтобы обеспечить необходимую точность, необходимо знать, какая кодировка применялась при создании двоичного представления:

```
>>> s
'AÄBäC'
>>> x = s.encode('utf-8').decode('utf-8') # OK: кодировки совпадают
>>> x
'AÄBäC'
>>> x = s.encode('latin-1').decode('latin-1') # можно использовать любые
>>> x                                         # совместимые кодировки
'AÄBäC'

>>> x = s.encode('utf-8').decode('latin-1')   # декодирование выполняется,
>>> x                                         # но получается мусор
UnicodeEncodeError: 'charmap' codec can't encode character '\xc3' in position 2:...

>>> len(s), len(x)                            # уже не та же самая строка
(5, 7)

>>> s.encode('utf-8')                         # не те же самые кодовые пункты
b'A\xc3\x84B\xc3\xa4C'
>>> x.encode('utf-8')
b'A\xc3\x83\xc2\x84B\xc3\x83\xc2\xa4C'
```

```
>>> s.encode('latin-1')
b'A\xc4B\xe4C'
>>> x.encode('latin-1')
b'A\xc3\x84B\xc3\xa4C'
```

Самое интересное, что после применения несовпадающих кодировок иногда оригинальную строку все еще можно восстановить. Если снова закодировать полученный мусор в кодировку Latin-1 (8-битовые символы) и затем декодировать с применением корректной кодировки, оригинальная строка будет восстановлена (благодаря этому обстоятельству в некоторых ситуациях вы можете все поправить, если при первой попытке данные были декодированы неправильно):

```
>>> s
'AÄBäC'
>>> s.encode('utf-8').decode('latin-1')
UnicodeEncodeError: 'charmap' codec can't encode character '\xc3' in position 2:...
>>> s.encode('utf-8').decode('latin-1').encode('latin-1')
b'A\xc3\x84B\xc3\xa4C'
>>> s.encode('utf-8').decode('latin-1').encode('latin-1').decode('utf-8')
'AÄBäC'
>>> s.encode('utf-8').decode('latin-1').encode('latin-1').decode('utf-8') == s
True
```

С другой стороны, для декодирования данных можно использовать различные кодировки, при условии, что они совместимы с кодировкой данных – при использовании кодировок ASCII, UTF-8 и Latin-1, например, операция декодирования текста ASCII возвращает один и тот же результат:

```
>>> 'spam'.encode('utf8').decode('latin1')
'spam'
>>> 'spam'.encode('latin1').decode('ascii')
'spam'
```

Важно помнить, что декодированная строка никак не зависит от кодировки, с применением которой она была получена. После декодирования понятие кодировки не может применяться к строке – она является обычной последовательностью символов Юникода («кодовых пунктов»). Таким образом, заботиться о кодировках необходимо только в точках передачи данных между программой и файлами:

```
>>> s
'AÄBäC'
>>> s.encode('utf-16').decode('utf-16') == s.encode('latin-1').decode('latin-1')
True
```

Текст Юникода в файлах

Те же правила действуют и для текстовых файлов, потому что строки Юникода сохраняются в файлах в виде декодированных байтов. При записи мы можем закодировать строку с применением любой кодиров-

ки, совместимой с символами, имеющимися в строке. Однако при чтении необходимо заранее знать кодировку или использовать ту, которая декодирует байты в те же самые символы:

```
>>> open('ldata', 'w', encoding='latin-1').write(s) # сохранить в latin-1
5
>>> open('udata', 'w', encoding='utf-8').write(s)   # сохранить в utf-8
5

>>> open('ldata', 'r', encoding='latin-1').read() # OK: корректное имя
'AÄBäC'
>>> open('udata', 'r', encoding='utf-8').read()
'AÄBäC'

>>> open('ldata', 'r').read()                     # иначе может вернуть ошибку
'AÄBäC'
>>> open('udata', 'r').read()
UnicodeEncodeError: 'charmap' codec can't encode characters in position 2-3: cha...

>>> open('ldata', 'r', encoding='utf-8').read()
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: invalid dat...
>>> open('udata', 'r', encoding='latin-1').read()
UnicodeEncodeError: 'charmap' codec can't encode character '\xc3' in position 2:...
```

Напротив, при чтении из файлов в двоичном режиме попытка декодировать данные в строку Юникода не производится. Они благополучно будут прочитаны независимо от того, в каком режиме были записаны эти данные – в текстовом, с автоматическим кодированием строк `str` (как в предыдущем интерактивном сеансе), или в двоичном, в виде строк `bytes`, закодированных вручную:

```
>>> open('ldata', 'rb').read()
b'A\xc4B\xe4C'
>>> open('udata', 'rb').read()
b'A\xc3\x84B\xc3\xa4C'

>>> open('sdata', 'wb').write( s.encode('utf-16') ) # вернет число: 12
>>> open('sdata', 'rb').read()
b'\xff\xfeA\x00\xc4\x00B\x00\xe4\x00C\x00'
```

Юникод и виджет Text

Все вышеизложенное имеет прямое отношение к виджету `Text`: если файл открывается в двоичном режиме, отпадает необходимость беспокоиться о кодировках – библиотека tkinter будет интерпретировать данные в соответствии с нашими ожиданиями, по крайней мере, для этих двух кодировок:

```
>>> from tkinter import Text
>>> t = Text()
>>> t.insert('1.0', open('ldata', 'rb').read())
>>> t.pack()                        # строка появится в графическом интерфейсе
```

```
>>> t.get('1.0', 'end')
'AÄBäC\n'
>>>
>>> t = Text()
>>> t.insert('1.0', open('udata', 'rb').read())
>>> t.pack()                        # строка появится в графическом интерфейсе
>>> t.get('1.0', 'end')
'AÄBäC\n'
```

Виджет действует, как если бы мы передали ему строку `str`, извлеченную в текстовом режиме, но при использовании текстового режима нам необходимо передать интерпретатору Python имя кодировки – операции чтения будут терпеть неудачу при использовании кодировки, несовместимой с данными, хранящимися в файле:

```
>>> t = Text()
>>> t.insert('1.0', open('ldata', 'r', encoding='latin-1').read())
>>> t.pack()
>>> t.get('1.0', 'end')
'AÄBäC\n'
>>>
>>> t = Text()
>>> t.insert('1.0', open('udata', 'r', encoding='utf-8').read())
>>> t.pack()
>>> t.get('1.0', 'end')
'AÄBäC\n'
```

В любом случае содержимое, извлеченное из файла, всегда будет представлено строкой Юникода `str`, поэтому двоичный режим оказывает влияние только на операцию чтения. Но нам тем не менее необходимо знать кодировку, сохраняем мы данные непосредственно в текстовом режиме или выполняем запись в двоичном режиме после кодирования вручную:

```
>>> c = t.get('1.0', 'end')
>>> c                               # содержимое - строка str
'AÄBäC\n'

>>> open('cdata', 'wb').write(c)   # binary mode needs bytes
TypeError: must be bytes or buffer, not str

>>> open('cdata', 'w', encoding='latin-1').write(c) # каждая операция записи
>>> open('cdata', 'rb').read()                      # возвращает число 6
b'A\xc4B\xe4C\r\n'

>>> open('cdata', 'w', encoding='utf-8').write(c)   # другие байты в файле
>>> open('cdata', 'rb').read()
b'A\xc3\x84B\xc3\xa4C\r\n'

>>> open('cdata', 'w', encoding='utf-16').write(c)
>>> open('cdata', 'rb').read()
```

```
b'\xff\xfeA\x00\xc4\x00B\x00\xe4\x00C\x00\r\x00\n\x00'

>>> open('cdata', 'wb').write( c.encode('latin-1') ) # закодировать вручную
>>> open('cdata', 'rb').read()                       # то же, но с \r в Win
b'A\xc4B\xe4C\n'

>>> open('cdata', 'w', encoding='ascii').write(c)    # должна быть совместимой
UnicodeEncodeError: 'ascii' codec can't encode character '\xc4' in position 1: o
```

Обратите внимание на последнюю операцию в этом примере: операция кодирования вручную и записи в файл может терпеть неудачу, если данные не смогут быть закодированы с применением указанной кодировки. Когда такое случается, программа, вероятно, должна восстановиться после исключения и попробовать альтернативную кодировку – это особенно справедливо для платформ, где в качестве кодировки по умолчанию используется ASCII.

Проблемы обработки текста, представленного байтами

Правила, описанные в предыдущем разделе, могут показаться слишком сложными, но в действительности все они сводятся к следующему:

- Кодировку, используемую в операциях чтения/записи в текстовом режиме и при кодировании/декодировании вручную, необходимо знать только при работе со строками, кодировка символов в которых не совпадает с кодировкой, используемой на текущей платформе по умолчанию.
- Для записи в новые файлы можно использовать практически любую кодировку, при условии, что она может применяться к символам в строках, но при чтении необходимо указывать кодировку, совместимую с имеющимися двоичными данными.
- При чтении текста в двоичном режиме для отображения в виджете знать кодировку необязательно, но виджет `Text` всегда возвращает содержимое в виде строки `str`, поэтому, чтобы закодировать его перед сохранением в файл, необходимо знать кодировку.

Так почему бы всегда не загружать текст для отображения в виджете `Text` в двоичном режиме? Хотя чтение из файлов в двоичном режиме и кажется решением проблем кодирования, тем не менее передача текста библиотеке tkinter в виде строк `bytes` вместо `str` в действительности просто перекладывает проблему кодирования на библиотеку Tk, которая налагает собственные ограничения.

В частности, может показаться, что открывая входные файлы в двоичном режиме, мы обеспечиваем возможность отображения произвольных текстовых данных, но у этого решения есть два недостатка:

- Оно снимает бремя выбора кодировки с нашего сценария и перекладывает его на библиотеку Tk. Библиотеке все равно придется решать,

как отображать полученные байты, и может так случиться, что она не будет поддерживать какие-то необходимые кодировки.

- Оно позволяет открывать и просматривать файлы, которые по своей природе не являются текстовыми, сводя к нулю некоторые из преимуществ, предлагаемых проверками, выполняемыми при декодировании текста.

Первый пункт является, пожалуй, наиболее важным. Поэкспериментировав в Windows, я выяснил, что библиотека Tk корректно обрабатывает строки `bytes` в кодировках ASCII, UTF-8 и Latin-1, но не справилась с кодировкой UTF-16 и другими, такими как CP500. Однако все эти строки отображаются корректно, когда перед передачей библиотеке Tk двоичные данные декодируются в строку `str`. В программах, предназначенных для использования по всему миру, такая расширенная поддержка становится жизненно значимой. Если у вас есть возможность определить кодировку или запросить ее у пользователя, то для отображения и сохранения текста в файлы лучше использовать строки `str`.

Независимо от того, передаете вы текстовые данные в виде строк типа `str` или `bytes`, графические интерфейсы на основе библиотеки tkinter подчиняются ограничениям, накладываемым библиотекой Tk и языком программирования Tcl, а также всеми приемами использования библиотеки tkinter в языке Python, которая служит интерфейсом к библиотеке Tk. Например:

- Tcl, внутренний язык реализации библиотеки Tk, хранит строки в кодировке UTF-8 и требует, чтобы строки передавались через его прикладной интерфейс на языке C именно в этом формате.
- Tcl пытается преобразовать строки байтов, используя кодировку UTF-8, и в целом поддерживает преобразования с использованием кодировок, определяемых региональными настройками системы и кодировки Latin-1, как последнего средства.
- Реализация библиотеки tkinter на языке Python передает строки `bytes` языку Tcl без промежуточных преобразований, но при использовании строк Юникода типа `str` копируются объекты Юникода языка Tcl.
- Библиотека Tk унаследовала все ограничения языка Tcl, связанные с Юникодом, и добавляет свои ограничения, связанные с выбором шрифта для отображения.

Иными словами, графические интерфейсы, отображающие текст с использованием средств библиотеки tkinter, находятся во власти нескольких слоев программного обеспечения, расположенных выше и ниже программного кода на языке Python. Но, как бы то ни было, Юникод достаточно полно поддерживается виджетом `Text` из библиотеки Tk при использовании строк типа `str`, хотя это не относится к строкам `bytes`. Как вы уже наверняка заметили, обсуждение этой проблемы быстро начинает обрастать техническими деталями, поэтому мы не будем ис-

следовать ее дальше в этой книге – дополнительные сведения о tkinter, Tk и Tcl и интерфейсах между ними ищите в Сети или в других источниках информации.

Другие проблемы двоичного режима

Даже в ситуациях, когда достаточно использовать файлы, открытые в двоичном режиме, обойти проблемы с кодировками оказывается сложнее, чем можно было бы подумать. При записи в двоичном режиме нам всегда придется проявлять осторожность, чтобы прочитанные данные позднее были корректно записаны в файл, – при чтении в двоичном режиме строки в Windows будут завершаться последовательностью символов \r\n и было бы нежелательно, чтобы при записи в текстовом режиме они превращались в последовательности \r\r\n. Кроме того, между строками типа str и bytes в tkinter существует еще одно отличие. Строки str, прочитанные из файла в текстовом режиме, выводятся в графическом интерфейсе в ожидаемом виде, и в Windows символы конца строки отображаются должным образом:

```
C:\...\PP4E\Gui\Tour> python
>>> from tkinter import *
>>> T = Text()                                # str в текстовом режиме
>>> T.insert('1.0', open('jack.txt').read()) # кодировка по умолчанию
>>> T.pack()                                  # нормально отображается в GUI
>>> T.get('1.0', 'end')[:75]
'000) All work and no play makes Jack a dull boy.\n001) All work and no pla'
```

Однако если передать в графический интерфейс строку bytes, прочитанную из файла в двоичном режиме, в Windows она будет выглядеть на экране довольно странно – в конце каждой строки текста появится лишний пробел, соответствующий символу \r, который не отсекается при чтении из файлов в двоичном режиме:

```
C:\...\PP4E\Gui\Tour> python
>>> from tkinter import *
>>> T = Text()                                      # bytes в двоичном режиме
>>> T.insert('1.0', open('jack.txt', 'rb').read()) # без декодирования
>>> T.pack()                                        # появились пробелы в конце
>>> T.get('1.0', 'end')[:75]                        # строк!
'000) All work and no play makes Jack a dull boy.\r\n001) All work and no pl'
```

При использовании строк bytes, чтобы обеспечить отображение произвольного текста в ожидаемом виде, нам дополнительно придется вручную удалять символы \r в концах строк. Вследствие этого предполагается, что комбинация \r\n не имеет какого-то специального значения в схеме кодирования текста, хотя, если эта последовательность в данных не будет означать конец строки, такие данные с большой долей вероятности будут вызывать другие проблемы при отображении. В следующем фрагменте реализовано удаление лишних пробелов в концах строк – входной файл открывается в двоичном режиме, и при чтении недекодированных байтов вручную удаляются символы \r:

```
C:\...\PP4E\Gui\Tour> python
>>> from tkinter import *      # используется тип bytes, удаляются символы \r
>>> T = Text()
>>> data = open('jack.txt', 'rb').read()
>>> data = data.replace(b'\r\n', b'\n')
>>> T.insert('1.0', data)
>>> T.pack()
>>> T.get('1.0', 'end')[:75]
'000) All work and no play makes Jack a dull boy.\n001) All work and no pla'
```

Чтобы позднее сохранить это содержимое, можно либо добавить символы \r, при выполнении в Windows, вручную выполнить кодирование в тип bytes и сохранить данные в двоичном режиме; либо открыть файл в текстовом режиме, чтобы объект файла сам добавил символы \r, если это необходимо, выполнил кодирование и записал содержимое строки str. Второй путь, вероятно, более простой, так как он не требует беспокоиться о различиях между платформами.

Однако в любом случае мы вновь оказываемся лицом к лицу с проблемой кодирования – мы можем либо положиться на кодировку по умолчанию для текущей платформы, либо получить имя кодировки из пользовательского интерфейса. В следующем фрагменте, например, объект текстового файла сам преобразует символы конца строки и применяет кодировку по умолчанию для текущей платформы. Если бы было необходимо обеспечить поддержку произвольного текста Юникода или работоспособность сценария на платформах, где кодировка по умолчанию не соответствует отображаемым символам, мы могли бы передавать имя кодировки явно (операция извлечения среза, используемая здесь, имеет тот же эффект, что и применение спецификатора позиции «end-1c» в библиотеке Tk):

...продолжение предыдущего сеанса...

```
>>> content = T.get('1.0', 'end')[:-1]       # отбросит \n в конце
>>> open('copyjack.txt', 'w').write(content) # кодировка по умолчанию
12500                                        # текстовый режим в Win добавит \n
>>> ^Z

C:\...\PP4E\Gui\Tour> fc jack.txt copyjack.txt
Comparing files jack.txt and COPYJACK.TXT
FC: no differences encountered
```

Поддержка Юникода в PyEdit (забегая вперед)

Пример использования поддержки Юникода в виджете Text мы увидим в главе 11, когда будем разбирать реализацию приложения PyEdit. В действительности под поддержкой Юникода подразумевается лишь поддержка различных кодировок при работе с файлами, открытыми в текстовом режиме, – как только текст окажется в памяти, его обработка всегда выполняется в терминах типа str, потому что библиотека tkinter возвращает содержимое именно в таком виде. Чтобы обеспечить

поддержку Юникода, редактор PyEdit открывает файлы для чтения и записи в текстовом режиме, явно указывая кодировку, если это возможно, а двоичный режим использует только как последнее средство. Благодаря этому отпадает необходимость полагаться на ограниченную поддержку Юникода в библиотеке Tk, предусмотренную для отображения строк байтов.

Для этого редактор PyEdit реализует возможность получения имен кодировок из самых разных источников и позволяет пользователям указывать, какие из них желательно использовать. Кодировка может быть получена в результате диалога с пользователем, из параметров настройки в конфигурационных файлах, из настроек системы по умолчанию, из значения, сохраненного ранее в файле, и даже из внутренних значений в программе (полученных в результате анализа заголовков сообщений электронной почты, например). Все эти источники опробуются друг за другом, пока не встретится первая подходящая кодировка, при этом в некоторых ситуациях может потребоваться ограничиться единственным источником.

Эту реализацию вы увидите а главе 14. Честно признаться, версия редактора PyEdit в этом издании изначально предусматривала чтение и запись в файлы в текстовом режиме с использованием кодировки по умолчанию. Я не предполагал заострять внимание на поддержке Юникода в PyEdit, пока не столкнулся с необходимостью поддержки самых разнообразных кодировок, существующих в Интернете, при подготовке примера PyMailGUI. Если вы считаете, что строки стали намного сложнее, чем могли бы быть, то это скорее всего, потому, что спектр ваших представлений остается слишком узким.

Более сложные операции с текстом и тегами

Но достаточно разговоров о сложностях Юникода – вернемся к программированию графических интерфейсов. Помимо указания позиции, теги виджета `Text` могут также использоваться для форматирования и выполнения операций как над всеми символами подстроки, так и над всеми подстроками, помещенными в тег. Эта особенность составляет значительную часть мощи, предоставляемой виджетом `Text`:

- Теги имеют атрибуты форматирования, позволяющие определять цвет, шрифт, ширину табуляции, величину межстрочных интервалов и параметры выравнивания. Они могут применяться одновременно к нескольким фрагментам текста, если связать их с тегом и выполнить настройку тега с помощью метода `tag_config`, который очень напоминает общий метод `config` виджетов.
- Теги позволяют выполнять привязку событий, что дает возможность реализовать, например, гиперссылки в виджете `Text`: щелчок на тексте вызывает обработчик события его тега. Привязка событий к тегам осуществляется с помощью метода `tag_bind`, во многом подобного уже знакомому общему методу `bind` виджетов.

С помощью тегов можно отображать текст с различными настройками внутри одного и того же виджета Text; например, ко всему виджету Text можно применить один шрифт, а к тексту в тегах – другие шрифты. Кроме того, виджет Text позволяет встраивать другие виджеты в заданное индексом место (они интерпретируются, как отдельный символ), а также графические изображения.

Пример 9.12 иллюстрирует основы применения сразу всех этих дополнительных возможностей и воспроизводит интерфейс, изображенный на рис. 9.22. Этот сценарий применяет форматирование и выполняет привязку событий к трем подстрокам, помеченным тегами, выводит текст с помощью двух разных комбинаций шрифтов и цветов, а также встраивает графическое изображение и кнопку. Двойной щелчок мышью на любой из подстрок, заключенных в теги (или на встроенной кнопке), генерирует событие, которое выводит в поток stdout сообщение «Got tag event».

Пример 9.12. PP4E\Gui\Tour\texttags.py

```
"демонстрация дополнительных возможностей тегов и виджета Text"

from tkinter import *
root = Tk()
def hello(event): print('Got tag event')

# создать и настроить виджет Text
text = Text()
text.config(font=('courier', 15, 'normal'))                # общий шрифт
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 строк

# 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')                     # добавить 'is' в тег
text.tag_add('demo', '3.0', '3.3')                     # добавить 'the' в тег
text.tag_add('demo', '5.3', '5.7')                     # добавить 'life' в тег
text.tag_config('demo', background='purple')           # изменить цвета тега
text.tag_config('demo', foreground='white')            # называются не bg/fg
text.tag_config('demo', font=('times', 16, 'underline')) # изменить шрифт тега
text.tag_bind('demo', '<Double-1>', hello)             # привязать события
root.mainloop()
```

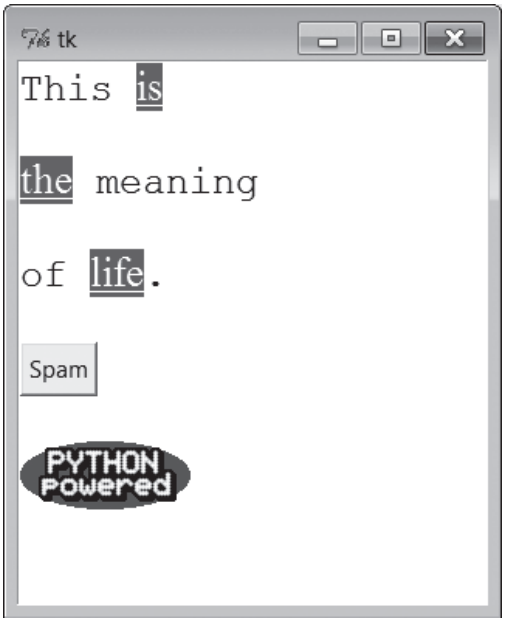


Рис. 9.22. *Теги виджета Text в действии*

Такие средства встраивания и работы с тегами тегов можно в конечном итоге использовать для отображения веб-страницы. А стандартный модуль `html.parser` анализа разметки HTML может помочь с автоматизацией построения графического интерфейса веб-страницы. Как можно догадаться, виджет `Text` предоставляет больше возможностей программирования графических интерфейсов, чем позволяет описать объем книги. За подробностями о возможностях, предоставляемых тегами и виджетом `Text`, обращайтесь к другим справочникам по библиотекам Tk и tkinter. А сейчас начнутся занятия в художественной школе.

Виджет Canvas

Что касается графики, то виджет `Canvas` (холст) из tkinter является самым свободным по форме инструментом в этой библиотеке. Он позволяет рисовать фигуры, динамически перемещать объекты и располагать другие виджеты. Холст основан на структурированной модели графического объекта: все, что изображается на холсте, может обрабатываться как *объект*. Можно опуститься до уровня обработки пикселей, а можно оперировать более крупными объектами, такими как фигуры, графические изображения и встроенные виджеты. Все это делает холст достаточно мощным инструментом, как для использования в простых графических редакторах, так и в полноценных программах визуализации и воспроизведения анимации.

Базовые операции с виджетом Canvas

Холсты повсеместно используются во многих нетривиальных графических интерфейсах, и далее в этой книге можно будет увидеть более крупные примеры использования холстов, в программах PyDraw, PyPhoto, PyView, PyClock и PyTree. А сейчас сразу займемся примером, в котором демонстрируются основы его применения. В примере 9.13 используется большинство основных методов создания изображений на холсте.

Пример 9.13. PP4E\Gui\Tour\canvas1.py

```
"демонстрация основных возможностей холста"

from tkinter import *

canvas = Canvas(width=525, height=300, bg='white') # 0,0 - верхний левый угол
canvas.pack(expand=YES, fill=BOTH)                 # рост вниз и вправо

canvas.create_line(100, 100, 200, 200)             # fromX, fromY, toX, toY
canvas.create_line(100, 200, 200, 300)             # рисование фигур
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/ora-lp4e.gif')
canvas.create_image(325, 25, image=photo, anchor=NW) # встроить изображение

widget = Label(canvas, text='Spam', fg='white', bg='black')
widget.pack()
canvas.create_window(100, 100, window=widget)        # встроить виджет
canvas.create_text(100, 280, text='Ham')             # нарисовать текст
mainloop()
```

Запущенный сценарий создаст окно, изображенное на рис. 9.23. Ранее уже было показано, как поместить на холст графическое изображение и установить соответствующие ему размеры холста (раздел «Изображения» в конце главы 8). Этот сценарий изображает также фигуры, текст и даже встроенный виджет `Label`. Его окно пока ограничивается только отображением – чуть ниже будет показано, как добавить обработчики событий, дающие пользователю возможность взаимодействовать с отображаемыми элементами.

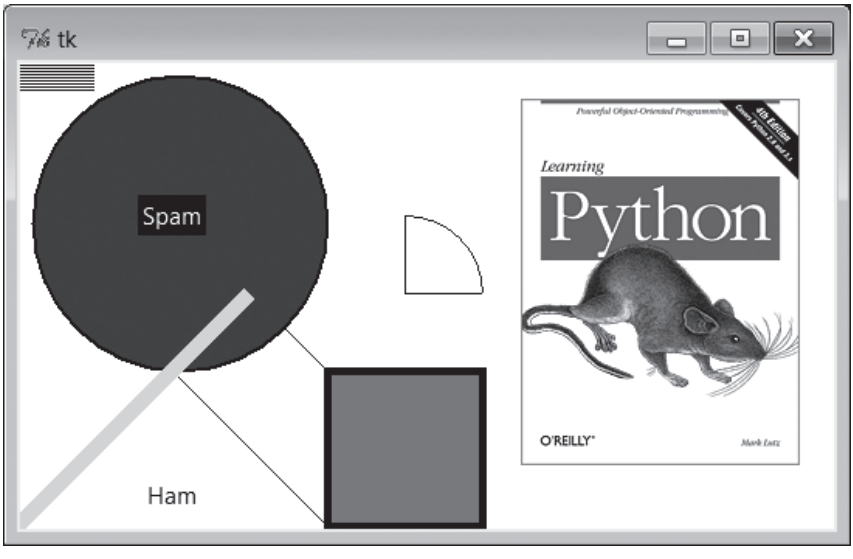


Рис. 9.23. *Окно сценария canvas1 с изображениями объектов*

Программирование виджета Canvas

Холсты просты в использовании, при этом они имеют свою систему координат, определяют уникальные методы рисования и именуют объекты с помощью идентификаторов или тегов. Данный раздел знакомит с этими базовыми понятиями холста.

Координаты

Все элементы, отображаемые на холсте, представляют собой отдельные объекты, которые в действительности не являются виджетами. Если внимательно изучить сценарий `canvas1`, можно заметить, что холсты создаются и прикрепляются к родительскому контейнеру, как и любые другие виджеты tkinter. Однако к предметам, изображаемым на холсте, это не относится: фигуры, графические изображения и другие элементы располагаются и перемещаются по холсту с помощью координат, идентификаторов и тегов. При этом координаты являются наиболее фундаментальной составляющей модели холста.

Холст определяет собственную систему координат (X,Y) для своей области отображения; X обозначает горизонтальную ось, Y – вертикальную. По умолчанию координаты измеряются в пикселях (точках); левый верхний угол холста имеет координаты (0,0), а координаты X и Y возрастают вправо и вниз соответственно. Чтобы нарисовать или разместить объект на холсте, необходимо указать одну или более пар ко-

ординат (X,Y), определяющих абсолютные местоположения на холсте. Такой способ отличается от ограничений, использовавшихся до сих пор для прикрепления виджетов, но он позволяет управлять графической структурой с очень большой точностью и поддерживает более свободные по форме технологии, например, анимацию.[1]

Создание объектов

Холст позволяет рисовать и отображать простые фигуры, такие как линии, овалы, прямоугольники, дуги и многоугольники. Кроме того, имеется возможность встраивать текст, графические изображения и другие виджеты tkinter, такие как метки и кнопки. В сценарии `canvas1` продемонстрированы все основные методы конструирования графических объектов – каждому из них передается один или более наборов координат (X,Y), определяющих координаты 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) # противоположные углы овала
id = canvas.create_arc( fromX, fromY, toX, toY) # противоположные углы дуги
id = canvas.create_rectangle(fromX, fromY, toX, toY) # противоположные углы
                                                     # прямоугольника
```

В других методах рисования указывается только одна пара координат (X,Y), определяющая координаты левого верхнего угла объекта:

```
id = canvas.create_image(250, 0, image=photo, anchor=NW) # встроить изображ.
id = canvas.create_window(100, 100, window=widget)       # встроить виджет
id = canvas.create_text(100, 280, text='Ham')            # нарисовать текст
```

Холст также предоставляет метод `create_polygon`, принимающий произвольное множество аргументов координат, определяющих точки, соединенные линиями. Его удобно использовать для рисования произвольных фигур, образованных отрезками прямых линий.

Помимо координат большинство методов рисования позволяет определять обычные параметры настройки, такие как ширина границы (`width`), цвет заливки (`fill`), цвет границы (`outline`) и так далее. У некоторых типов объектов есть собственные уникальные параметры настройки; например, для линий можно указать форму необязательной стрелки,

---

[1] О технике анимации рассказывается в конце данного обзора. Поскольку в область отрисовки холста можно встраивать другие виджеты, собственная координатная система делает холсты идеальными инструментами для реализации графических интерфейсов, дающих пользователям возможность создания других графических интерфейсов путем перетаскивания встроенных виджетов по холсту – весьма полезное применение холста, которое мы изучили бы в этой книге, будь у меня несколько сотен лишних страниц.

а текст, виджеты и изображения можно привязывать по направлениям сторон света (что похоже на параметр anchor менеджера компоновки, но в действительности определяет точку объекта, помещаемую в координаты (X,Y), указанные в вызове метода create; NW, например, помещает в координаты (X,Y) левый верхний угол объекта).

Важнее всего, вероятно, отметить здесь, что библиотека tkinter по большей части сама выполняет «черновую» работу – рисуя фигуры, вы только указываете координаты, а библиотека сама вычерчивает и отображает их. Если вам когда-либо приходилось заниматься графикой на низком уровне, то вы оцените разницу.

Идентификаторы объектов и операции

Сценарий canvas1 не использует тот факт, что у каждого помещаемого на холст объекта есть идентификатор. Его возвращает метод create_, который рисует или встраивает объект (в примерах предыдущего раздела он был представлен переменной id). Этот идентификатор можно впоследствии передавать другим методам, чтобы переместить объект в новые координаты, установить параметры его настройки, удалить с холста, поднять или опустить относительно других перекрывающихся объектов и так далее

Например, метод move холста может принимать идентификатор объекта и смещения (не координаты) X и Y, и перемещать объект согласно заданному смещению:

```
canvas.move(objectIdOrTag, offsetX, offsetY) # переместить объект(ы)
```

Если при этом объект смещается за пределы холста, он просто обрезается (не показывается). К объектам можно также применять другие часто используемые операции:

```
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 языке 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 библиотека tkinter оперирует структурированными объектами – не нужно поднимать и перерисовывать каждый пиксель вручную, чтобы осуществить перемещение или подъем объекта.

Теги объектов в виджете Canvas

Однако холсты способны предложить еще более мощные возможности, чем мы видели до сих пор. Помимо идентификаторов объектов существуют теги. *Тег* – создаваемое программистом имя, с которым можно связать ряд отображаемых объектов и применить ту или иную операцию сразу ко всей группе. Пометка объектов тегами в виджете `Canvas` по крайней мере по духу сходна с пометкой тегами подстрок в виджете `Text`, рассматривавшемся 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 методу `move` холста. Именно по этой причине метод `move` принимает смещения, а не координаты – получая тег, он перемещает каждый ассоциированный с этим тегом объект на указанные смещения (X,Y); если бы метод принимал абсолютные координаты, все связанные с тегом объекты могли бы оказаться в одном и том же месте друг над другом.

Чтобы связать объект с тегом, нужно указать имя тега в параметре `tag` метода, 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)
```

Эти инструкции создают три овала и перемещают 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 настраивать и обрабатывать независимо.

Как и виджет `Text`, виджет `Canvas` имеет теги с предопределенными именами: тег `all` ссылается на все объекты, имеющиеся на холсте, а `current` ссылается на тот объект, который находится под указателем мыши. Можно не только запрашивать идентификатор объекта под указателем мыши, но и осуществлять поиск объектов с помощью методов `find_` холста: например, метод `canvas.find_closest(X,Y)` возвращает кортеж, первый элемент которого содержит идентификатор объекта, находящегося ближе всего к точке с указанными координатами, – это удобно, когда уже есть координаты, полученные в обработчике события, сгенерированного щелчком мыши.

К представлению о тегах холста мы вернемся снова в более позднем примере из этой главы (если вам нужны подробности прямо сейчас, можете посмотреть сценарии воспроизведения анимации ближе к концу). Холсты поддерживают другие операции и параметры, для рассказа о кото-

рых здесь недостаточно места (например, метод холста `postscript` позволяет сохранить холст в файле в формате PostScript). Дополнительные сведения можно найти в примерах, имеющихся далее в книге, таких как PyDraw, а полный список параметров объекта холста можно найти в справочниках по библиотеке Tk или tkinter.

Прокрутка холстов

Однако одна из операций над холстами настолько часто используется в практике, что действительно заслуживает внимания. Как демонстрирует пример 9.14, полосы прокрутки можно перекрестно связывать с холстами, используя те же протоколы, которые ранее использовались для добавления их к виджетам `Listbox` и `Text`, но с некоторыми особыми требованиями.

Пример 9.14. PP4E\Gui\Tour\scrolledcanvas.py

```
"простой компонент холста с вертикальной прокруткой"

from tkinter import *

class ScrolledCanvas(Frame):
    def __init__(self, parent=None, color='brown'):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)            # сделать растягиваемым
        canv = Canvas(self, bg=color, relief=SUNKEN)
        canv.config(width=300, height=200)          # размер видимой области
        canv.config(scrollregion=(0, 0, 300, 1000))  # углы холста
        canv.config(highlightthickness=0)           # без рамки

        sbar = Scrollbar(self)
        sbar.config(command=canv.yview)      # связать sbar и canv
        canv.config(yscrollcommand=sbar.set) # сдвиг одного = сдвиг другого
        sbar.pack(side=RIGHT, fill=Y)        # первым добавлен – посл. обрезан
        canv.pack(side=LEFT, expand=YES, fill=BOTH) # canv обрезается первым

        self.fillContent(canv)
        canv.bind('<Double-1>', self.onDoubleClick) # установить обр. события
        self.canvas = canv

    def fillContent(self, canv):                    # переопределить при
        for i in range(10):                         # наследовании
            canv.create_text(150, 50+(i*100), text='spam'+str(i),fill='beige')

    def onDoubleClick(self, event):                 # переопределить при
        print(event.x, event.y)                     # наследовании
        print(self.canvas.canvasx(event.x), self.canvas.canvasy(event.y))

if __name__ == '__main__': ScrolledCanvas().mainloop()
```

Этот сценарий создает окно, изображенное на рис. 9.24. Оно аналогично предыдущим примерам реализации прокрутки, но в модели прокрутки холстов есть две особенности:

Размеры области прокрутки и видимой области

Можно указать размер видимой области холста, но при этом обязательно следует указать и размер прокручиваемой области холста в целом. Видимая область – это область, отображаемая в окне, размер которой может изменяться при изменении размеров окна. Прокручиваемая область в общем случае должна быть больше – она включает все содержимое холста, из которого только часть отображается в видимой области. Операция прокрутки как бы перемещает окно просмотра по прокручиваемой области холста.

Отображение координат в области просмотра в абсолютные координаты

Кроме того, нужно устанавливать соответствие между координатами видимой области и координатами всего холста, если холст больше, чем видимая его область. Если холст поддерживает возможность прокрутки, он почти всегда больше видимой его области, и в этих случаях часто требуется вычислять это соответствие. В некоторых случаях необходимость в таком вычислении отсутствует, потому что виджеты, встраиваемые в холст, самостоятельно откликаются на действия пользователей (например, кнопки в примере PyPhoto, в главе 11). Однако если пользователь взаимодействует непосредственно с холстом (например, в графическом редакторе), преобразование из системы координат видимой области в систему координат всего холста становится насущной необходимостью.

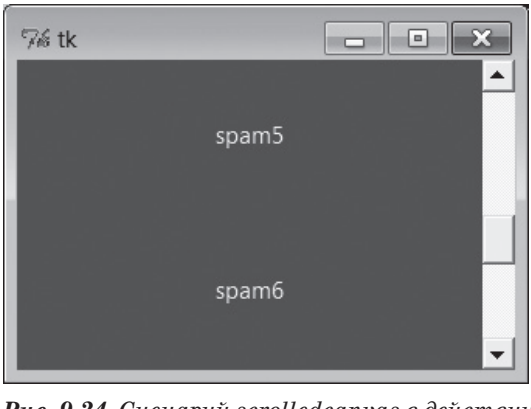


Рис. 9.24. *Сценарий scrolledcanvas в действии*

Размеры определяются как параметры настройки. Размер видимой области определяется с помощью параметров холста `width` и `height`. Чтобы определить общий размер холста, в параметре `scrollregion` следует

передать кортеж с четырьмя координатами верхнего левого и нижнего правого углов холста. Если размер видимой области не указан, используется размер по умолчанию. Если не задан параметр `scrollregion`, он принимается по умолчанию равным размеру видимой области. При этом полоса прокрутки становится бесполезной, так как видимая область в этом случае вмещает весь холст целиком.

Пересчет координат осуществляется несколько более сложным образом. Если прокручиваемая видимая область холста оказывается меньше, чем холст в целом, то координаты (X,Y), возвращаемые в объектах событий, представляют собой координаты в видимой области, а не в холсте в целом. Обычно требуется перевести координаты события в координаты холста, для чего они передаются методам `canvasx` и `canvasy` прежде чем их можно будет использовать для обработки объектов.

Например, если запустить сценарий реализации прокручиваемого холста и посмотреть на сообщения, выводимые при выполнении двойных щелчков мышью, можно заметить, что координаты события всегда относятся к видимой области, а не холсту в целом:

```
C:\...\PP4E\Gui\Tour> python scrolledcanvas.py
2 0             координаты x,y события, если холст прокручен доверху
2.0 0.0         координаты x,y холста – те же, если нет пикселей границы
150 106
150.0 106.0
299 197
299.0 197.0
3 2             координаты x,y события, если холст прокручен донизу
3.0 802.0       координаты x,y холста – координата y отличается
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 отображаемая координата X видимой области холста всегда совпадает с координатой X всего холста, поскольку видимая область и холст имеют одинаковую ширину 300 пикселей (из-за автоматической установки границ могло бы появиться расхождение в два пикселя, если бы не значение `highlightthickness`, установленное в сценарии). Обратите внимание, что после щелчка на вертикальной полосе прокрутки отображаемая координата Y видимой области становится отличной от координаты Y холста. Без преобразования координат значение координаты Y события неверно указывало бы на место, находящееся на холсте значительно выше.

В большинстве примеров с холстами в этой книге выполнять такое преобразование не требуется – координаты (0,0) всегда соответствует левому верхнему углу холста, в котором происходит щелчок мышью, но

лишь потому, что эти холсты не поддерживают прокрутку. Пример холста с двумя полосами прокрутки, горизонтальной и вертикальной, вы найдете в следующем разделе. Программа PyTree, что приводится далее в этой книге, также демонстрирует похожий холст, но при этом она динамически изменяет размеры области прокрутки при отображении новых деревьев.

Как правило, если холст поддерживает возможность прокрутки, то в обработчиках событий, учитывающих позицию, необходимо преобразовывать координаты события в действительные координаты холста. Для некоторых обработчиков это не требуется, если они привязаны к отдельным объектам или виджетам на холсте, а не к самому холсту; в следующих двух разделах показано, что происходит в этом случае.

Холсты с поддержкой прокрутки и миниатюр изображений

В конце главы 8 мы рассматривали коллекцию сценариев, отображающих миниатюры всех изображений, хранящихся в каталоге. Там отмечалось, что поддержка прокрутки является важным требованием для больших коллекций изображений. Теперь, когда мы знаем, как объединять холсты и полосы прокрутки, можно, наконец, задействовать их для реализации этого крайне необходимого расширения и закончить реализацию инструмента просмотра изображений, начатую в главе 8 (ну, или почти закончить).

Пример 9.15 представляет собой измененную версию последнего примера из предыдущей главы, которая отображает миниатюры в холсте с прокруткой. Описание особенностей функционирования сценария, а также модуля `ImageTk` (необходим для создания миниатюр и отображения изображений в формате JPEG), импортируемого из сторонней библиотеки Python Imaging Library (PIL), смотрите в предыдущей главе.

Фактически, чтобы полностью понять реализацию примера 9.15, необходимо также вспомнить пример 8.45, поскольку здесь мы повторно используем инструменты создания миниатюр и просмотра изображений из этого модуля. Здесь мы просто добавляем холст, позиционируем кнопки фиксированного размера по абсолютным координатам в холсте и вычисляем размеры прокручиваемой области, используя понятия, обозначенные в предыдущем разделе. Наличие горизонтальной и вертикальной полос прокрутки обеспечивает возможность свободного перемещения по всему холсту с кнопками независимо от их количества.

Пример 9.15. PP4E\Gui\PIL\viewer_thumbs_scrolled.py

```
"""
расширенная версия сценария просмотра изображений: отображает миниатюры на
кнопках фиксированного размера, чтобы обеспечить равномерное их размещение,
и добавляет возможность прокрутки при просмотре больших коллекций изображений,
отображая миниатюры в виджете Canvas с полосами прокрутки; требует наличия биб-
```

```
лиотеки PIL для отображения изображений в таких форматах, как JPEG, и повторно
использует инструменты создания миниатюр и просмотра единственного изображения из
сценария viewer_thumbs.py; предостережение/что сделать: можно также реализовать
возможность прокрутки при отображении единственного изображения, если его размеры
оказываются больше размеров экрана, которое сейчас обрезается в Windows; более
полная версия представлена в виде приложения PyPhoto в главе 11;
“””

import sys, math
from tkinter import *
from PIL.ImageTk import PhotoImage
from viewer_thumbs import makeThumbs, ViewOne

def viewer(imgdir, kind=Toplevel, numcols=None, height=300, width=300):
    “””
    использует кнопки фиксированного размера и холст с возможностью прокрутки;
    определяет размер области прокрутки (всего холста) и располагает
    миниатюры по абсолютным координатам x,y холста; предупреждение:
    предполагается, что все миниатюры имеют одинаковые размеры
    “””
    win = kind()
    win.title(‘Simple viewer: ‘ + imgdir)
    quit = Button(win, text='Quit', command=win.quit, bg='beige')
    quit.pack(side=BOTTOM, fill=X)

    canvas = Canvas(win, borderwidth=0)
    vbar = Scrollbar(win)
    hbar = Scrollbar(win, orient='horizontal')

    vbar.pack(side=RIGHT, fill=Y)    # прикрепить холст после полос прокрутки
    hbar.pack(side=BOTTOM, fill=X)   # чтобы он обрезался первым
    canvas.pack(side=TOP, fill=BOTH, expand=YES)

    vbar.config(command=canvas.yview)          # обработчики событий
    hbar.config(command=canvas.xview)          # перемещения полос прокрутки
    canvas.config(yscrollcommand=vbar.set)     # обработчики событий
    canvas.config(xscrollcommand=hbar.set)     # прокрутки холста
    canvas.config(height=height, width=width)  # начальные размеры видимой
                                               # области, изменяемой при
                                               # изменении размеров окна
    thumbs = makeThumbs(imgdir)                # [(imgfile, imgobj)]
    numthumbs = len(thumbs)
    if not numcols:
        numcols = int(math.ceil(math.sqrt(numthumbs))) # фиксиров. или N x N
    numrows = int(math.ceil(numthumbs / numcols))    # истинное деление в 3.x

    linksize = max(thumbs[0][1].size)              # (ширина, высота)
    fullsize = (0, 0,                              # верхний левый угол X,Y
        (linksize * numcols), (linksize * numrows) ) # нижний правый угол X,Y
    canvas.config(scrollregion=fullsize)           # размер области
                                                   # прокрутки
```

```
        rowpos = 0
        savephotos = []
        while thumbs:
            thumbsrow, thumbs = thumbs[:numcols], thumbs[numcols:]
            colpos = 0
            for (imgfile, imgobj) in thumbsrow:
                photo = PhotoImage(imgobj)
                link = Button(canvas, image=photo)
                handler = lambda savefile=imgfile: ViewOne(imgdir, savefile)
                link.config(command=handler, width=linksize, height=linksize)
                link.pack(side=LEFT, expand=YES)
                canvas.create_window(colpos, rowpos, anchor=NW,
                        window=link, width=linksize, height=linksize)
                colpos += linksize
                savephotos.append(photo)
            rowpos += linksize
        return win, savephotos

    if __name__ == '__main__':
        imgdir = 'images' if len(sys.argv) < 2 else sys.argv[1]
        main, save = viewer(imgdir, kind=Tk)
        main.mainloop()
```

Чтобы увидеть эту программу в действии, убедитесь, что у вас установлено расширение PIL, как описывается в главе 8, и запустите сценарий из командной строки, передав ему имя каталога с изображениями в виде аргумента:

```
...\PP4E\Gui\PIL> viewer_thumbs_scrolled.py C:\Users\mark\temp\101MSDCF
```

Как и прежде, щелчок на миниатюре открывает соответствующее полноразмерное изображение в новом окне. На рис. 9.25 изображено окно сценария при просмотре каталога с большим количеством цифровых фотографий. При первом запуске сценарий тратит значительное время на подготовку кэша с миниатюрами, зато последующие запуски выполняются быстро.

Можно также просто запустить сценарий без аргументов из командной строки щелчком на ярлыке файла или из среды IDLE. В этом случае он отобразит содержимое подкаталога с примерами изображений в дереве примеров для книги, как показано на рис. 9.26.

Еще о прокрутке изображений: PyPhoto (забегая вперед)

Несмотря на все эволюционные изменения, сценарий отображения миниатюр с прокруткой из примера 9.15 все еще имеет одно ограничение: изображения, размеры которых превышают размеры физического экрана, просто обрезаются при отображении в Windows. Этот недостаток наиболее очевидно проявляется при открытии больших цифровых фотографий, таких как на рис. 9.25. Кроме того, сценарий не позволяет изменять размеры уже открытых изображений, открывать другие ка-

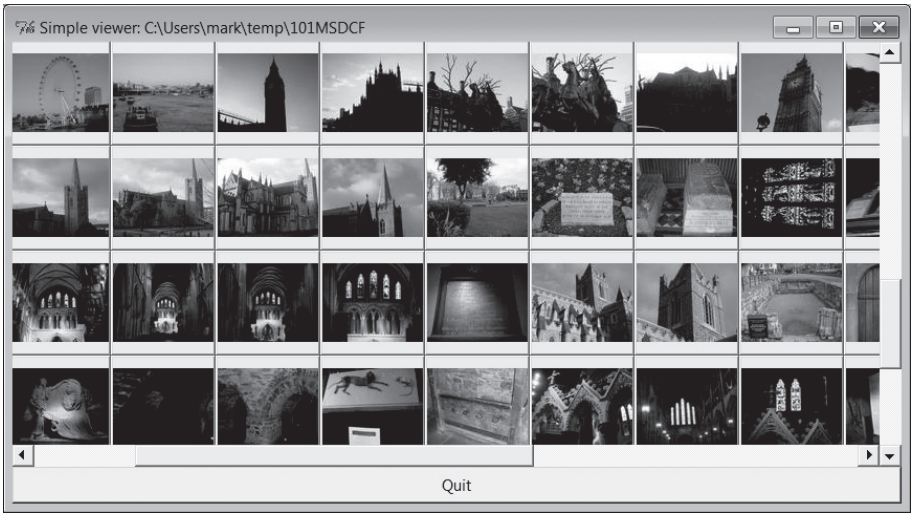


Рис. 9.25. *Сценарий отображения коллекции миниатюр с прокруткой*

Рис. 9.26. *Отображение каталога с изображениями по умолчанию*

талоги и так далее. Это довольно упрощенная демонстрация приемов программирования холста.

В главе 11 мы узнаем, как решить все эти проблемы, когда встретимся с программой PyPhoto. Эта программа добавляет возможность прокрутки и в окно просмотра полноразмерного изображения. Кроме того, она

позволяет изменять размеры изображения, поддерживает возможность сохранения изображений в файлы и открытия других каталогов в процессе выполнения. При этом в основе PyPhoto лежат те же приемы, которые были продемонстрированы в простом сценарии, представленном здесь, а также реализация генератора миниатюр, написанная в предыдущей главе. Как и простой текстовый редактор, демонстрировавшийся выше в этой главе, данная реализация является, по сути, прототипом более полнофункциональной программы PyPhoto, которую мы рассмотрим в главе 11. Оставайтесь с нами до волнующего финала истории развития PyPhoto (или, если ожидание для вас слишком томительно, шагайте вперед прямо сейчас).

Продолжая тему этой главы, обратите внимание, что операции просмотра изображений в примере 9.15 ассоциированы со встроенными виджетами кнопок, а не с самим холстом. Фактически холст здесь является не более, чем инструментом отображения. Чтобы увидеть, как обогатить его собственными событиями, перейдем к следующему разделу.

События холстов

Подобно виджетам `Text` и `Listbox` виджет `Canvas` не имеет параметра настройки `command` для назначения обработчика событий. Вместо этого программы, содержащие холсты, обычно используют другие виджеты (такие как кнопки с миниатюрами в примере 9.15) или низкоуровневый метод `bind`, чтобы установить обработчики щелчков мышью, нажатий клавиш и других событий (как в примере 9.14 реализации холста с прокруткой). Пример 9.16 берет за основу последний подход и демонстрирует, как привязать события самого холста, чтобы реализовать некоторые наиболее типичные операции рисования.

Пример 9.16. PP4E\Gui\Tour\canvasDraw.py

```
"""
реализует возможность рисования эластичных фигур на холсте при перемещении
указателя мыши с нажатой правой кнопкой; версии этого сценария, дополненные
тегами и анимацией, вы найдете в файлах canvasDraw_tags*.py
"""

from tkinter import *
trace = False

class CanvasEventsDemo:
    def __init__(self, parent=None):
        canvas = Canvas(width=300, height=300, bg='beige')
        canvas.pack()
        canvas.bind('<ButtonPress-1>', self.onStart) # щелчок
        canvas.bind('<B1-Motion>', self.onGrow)      # и вытягивание
        canvas.bind('<Double-1>', self.onClear)      # удалить все
        canvas.bind('<ButtonPress-3>', self.onMove)  # перемещать 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] # начало вытягивания
            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')                   # использовать тег all

        def onMove(self, event):
            if self.drawn:                               # передвинуть в позицию
                if trace: print(self.drawn)              # щелчка
                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, выполняемые мышью:

Очистка холста

Чтобы удалить все имеющееся на холсте, сценарий привязывает событие двойного щелчка левой кнопкой к методу `delete` холста с тегом `all` – встроенным тегом, который автоматически ассоциируется с каждым объектом на экране. Обратите внимание, что доступ к виджету холста, на котором выполнен щелчок, осуществляется через объект события, передаваемый обработчику (он также доступен через `self.canvas`).

Вытягивание фигур

При нажатии левой кнопки мыши и перетаскивании (перемещении при нажатой кнопке) создается прямоугольник или овал. Этот прием часто называется вытягиванием фигуры – фигура увеличивается и сжимается как резиновая, и окончательный ее размер и положение определяются позицией указателя мыши, где будет отпущена кнопка мыши.

Чтобы выполнить такую работу с помощью tkinter, нужно всякий раз, когда возникает событие перетаскивания, стирать старую фигуру и рисовать новую. Обе операции, удаления и рисования, выполняются настолько быстро, что возникает эффект эластичного вытягивания. Конечно, чтобы начертить фигуру в соответствии с текущим положением указателя 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 указателя мыши стирают прежнюю фигуру и рисуют новую от начальных координат до новых координат мыши, сохраняя идентификатор нового объекта для операции стирания в следующем событии.

Перемещение объектов

При щелчке правой кнопкой мыши (кнопкой 3) сценарий сразу перемещает объект, нарисованный последним, в то место, где произведен щелчок. Аргумент события `event` дает координаты (X,Y) щелчка, из которых вычитаются начальные координаты последнего нарисованного объекта, чтобы получить смещения (X,Y), передаваемые методу `move` холста (напомню, что метод `move` ожидает получить смещения, а не координаты). Не следует забывать о необходимости сначала пересчитать координаты события, если холст прокручен.

В итоге после взаимодействия с пользователем получается окно, подобное изображенному на рис. 9.27. При вытягивании объектов сценарий поочередно вычерчивает овалы и прямоугольники. Установите в сценарии глобальную переменную `trace` и вы увидите в `stdout` идентификаторы новых объектов, вычерчиваемых при вытягивании и перемещении. Этот снимок экрана получен после вытягивания и перемещения нескольких объектов, что невозможно понять, только глядя на снимок – запустите пример на своем компьютере, чтобы лучше ощутить выполняемые им операции.

Привязка событий к конкретным элементам

Подобно тому как мы делали это для виджета `Text`, мы можем привязать события к одному или нескольким конкретным объектам, нарисованным в виджете `Canvas`, с помощью его метода `tag_bind`. Этот метод принимает в качестве первого аргумента строку с именем тега или идентификатор объекта. Например, можно зарегистрировать отдельные обработчики событий щелчков мыши для каждого нарисованного элемента или для группы нарисованных и помеченных тегом элементов вместо обработчика для холста в целом. В примере 9.17 для иллюстрации выполняется привязка обработчика двойного щелчка как к самому холсту, так и к двум конкретным текстовым элементам на нем. Он создает окно, изображенное на рис. 9.28.

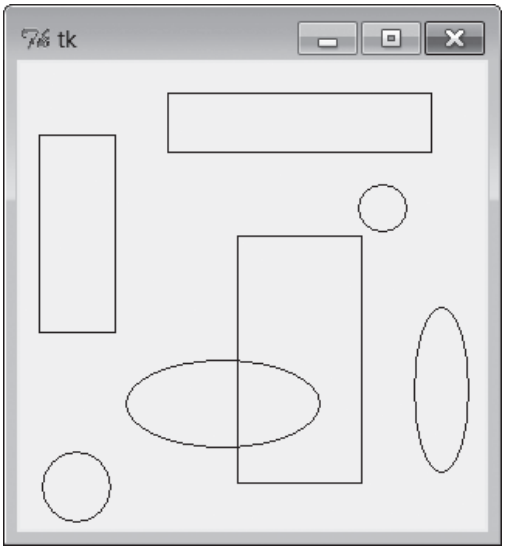


Рис. 9.27. *Окно сценария canvasDraw после нескольких вытягиваний и перемещений*

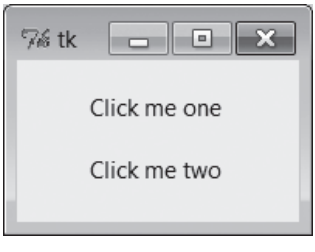


Рис. 9.28. *Окно сценария canvas-bind*

Пример 9.17. PP4E\Gui\Tour\canvas-bind.py

```
# привязка обработчиков событий к холсту и к элементам на нем
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, end=' ')
    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 самому холсту
canv.tag_bind(obj1, '<Double-1>', onObjectClick) # привязать к элементу
canv.tag_bind(obj2, '<Double-1>', onObjectClick) # теги тоже можно
canv.pack()                                      # использовать
root.mainloop()
```

Здесь методу `tag_bind` передаются идентификаторы объектов, но ему можно также передавать строку с именем тега, что позволяет привязывать обработчики событий к группам элементов. Когда щелчок выполняется в окне сценария за границами текстовых элементов, вызывается обработчик события холста. Когда щелчок выполняется на любом текстовом элементе, вызываются оба обработчика событий – холста и элемента. Ниже показан вывод в `stdout` после двух щелчков на холсте и одного щелчка на каждом из текстовых элементов. Чтобы получить идентификатор объекта для конкретного текстового элемента, ближайшего к точке щелчка, вызывается метод `find_closest` холста:

```
C:\...\PP4E\Gui\Tour> python canvas-bind.py
Got canvas click 3 6 .8217952        щелчки на холсте
Got canvas click 46 52 .8217952
Got object click 51 33 .8217952 (1,) щелчок на первом текстовом элементе
Got canvas click 51 33 .8217952
Got object click 55 69 .8217952 (2,) щелчок на втором текстовом элементе
Got canvas click 55 69 .8217952
```

Мы еще раз вернемся к идее событий, привязываемых к холсту, в примере PyDraw, в главе 11, где они будут использоваться для реализации полнофункционального графического редактора. Мы также вернемся к сценарию `canvasDraw` далее в этой главе, где добавим перемещения, основанные на тегах, и простую анимацию с применением инструментов, отмеряющих время, поэтому сделайте закладку на этой странице для справок в дальнейшем. Однако сначала давайте свернем немного в сторону и исследуем другой способ компоновки виджетов в окнах – модель компоновки по сетке.

Сетки

До сих пор мы размещали виджеты на экране, вызывая их метод `pack` – интерфейс к менеджеру компоновки в библиотеке tkinter. Мы также использовали абсолютные координаты в холстах, которые тоже можно считать своеобразным механизмом компоновки, хотя и не таким высокоуровневым, как методы менеджера компоновки. В данном разделе мы познакомимся с методом `grid`, наиболее часто используемой альтернативой методу `pack`. Мы предварительно уже рассматривали эту альтернативу в главе 8, когда обсуждали формы ввода и упорядочивали миниатюры изображений. А сейчас познакомимся с механизмом компоновки по сетке во всей полноте.

Как уже говорилось, действие менеджеров компоновки в библиотеке tkinter заключается в размещении дочерних виджетов внутри контейнера – родительского виджета (обычно родительскими являются элементы `Frame` или окна верхнего уровня). Когда виджету предлагается присоединить себя вызовом метода `pack` или `grid`, в действительности мы предлагаем его родительскому элементу расположить данный виджет среди собратьев. При использовании метода `pack` определяются некоторые ограничения, и менеджеру компоновки предлагается расположить виджеты соответствующим образом. При использовании метода `grid` виджеты располагаются в родительском элементе-контейнере по рядам и колонкам, как если бы он был таблицей.

Расположение по сетке является в библиотеке tkinter совершенно отдельной системой управления компоновкой. На момент написания этой книги методы `pack` и `grid` взаимно исключают друг друга; при расположении виджетов в одном и том же родительском элементе – в одном контейнере можно использовать либо метод `pack`, либо метод `grid`, но не тот и другой одновременно. Это разумно, если усвоить, что менеджеры компоновки выполняют свою работу в родительских элементах, и виджет может размещаться только одним менеджером компоновки.

В чем преимущества размещения по сетке?

Однако это означает, что, по крайней мере, внутри одного контейнера придется выбирать между методами `grid` и `pack` и придерживаться этого метода. Зачем тогда нужна сетка? В целом метод `grid` удобно использовать, когда необходимо расположить несвязанные между собой виджеты по рядам. Сюда относятся табличные интерфейсы и формы – расположение полей ввода по рядам и колонкам ничуть не сложнее, чем организация структуры интерфейса с помощью вложенных фреймов.

Как уже упоминалось в предыдущей главе, *формы ввода* выглядят привлекательнее, если виджеты располагаются по сетке или во фреймах-рядах с метками фиксированной длины, когда метки и поля ввода находятся на одной горизонтальной линии (как мы уже знаем, использование фреймов-колонок не обеспечивает необходимой точности расположения). Реализация структур интерфейсов на основе сеток и фреймов-рядов требуют примерно одинаковых усилий, тем не менее сетки удобнее, когда отсутствует возможность определить максимальную длину метки. Кроме того, сетки также могут использоваться для создания таблиц более сложного вида, чем формы.

Однако, как мы увидим, на практике метод `grid` не позволяет заметно уменьшить объем программного кода или его сложность, в сравнении с эквивалентными решениями на основе метода `pack`, особенно если в графическом интерфейсе должны решаться задачи изменения размеров. Иными словами, выбор между двумя схемами компоновки является в значительной мере вопросом стиля, а не техники.

Основы работы с сеткой: еще раз о формах ввода

Начнем с базовых понятий. В примере 9.18 создается таблица из меток и полей ввода – уже знакомых нам виджетов `Label` и `Entry`. Однако в данном случае они располагаются по сетке.

Пример 9.18. PP4E\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 += 1

mainloop()
```

Расположение по сетке заключается в назначении виджетам номеров рядов и колонок, отсчет которых начинается с 0, – библиотека tkinter использует эти координаты, а также размеры виджетов, чтобы расположить виджеты внутри контейнера. Это напоминает действие метода `pack`, только в данном случае понятия сторон и порядка прикрепления заменяются рядами и колонками.

Если запустить этот сценарий, то он создаст окно, изображенное на рис. 9.29 с данными, введенными в некоторые поля. И снова эта книга не позволяет увидеть цвета, отображаемые в правой части, поэтому вам придется включить свое воображение (или запустить сценарий на своем компьютере).

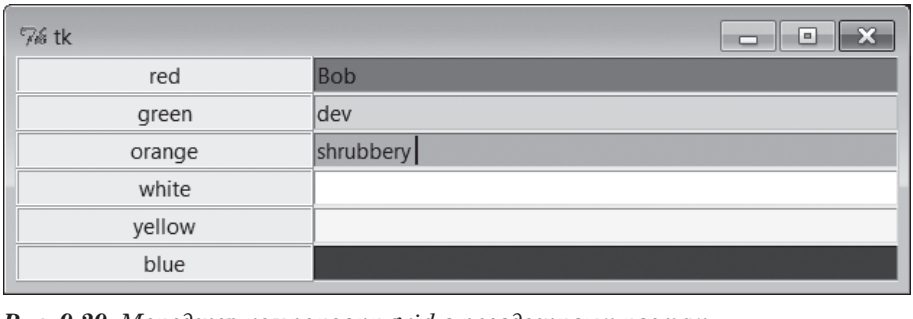


***Рис. 9.29.** Менеджер компоновки grid в псевдоживых цветах*

В действительности это та же самая классическая структура *формы ввода*, которую мы видели в предыдущей главе. Метки в левой части описывают данные, которые должны вводиться в поля справа.

Исключительно в целях демонстрации этот сценарий отображает слева названия цветов, в которые окрашены соответствующие поля ввода справа. Табличный вид достигается с помощью следующих строк:

```
Label(...).grid(row=r, column=0)
Entry(...).grid(row=r, column=1)
```

С точки зрения контейнера, метка помещается в колонку 0 строки с текущим номером (отсчет начинается с 0), а поле ввода помещается в колонку 1. В итоге система компоновки по сетке автоматически располагает метки и поля ввода в виде двухмерной таблицы, обеспечивая одинаковую высоту рядов и ширину колонок, достаточные для размещения самого большого элемента в каждой колонке.

То есть благодаря размещению виджетов по рядам и колонкам они должным образом выравниваются по обоим направлениям. Использование метода `pack` с фреймами-рядами позволяет добиться того же эффекта, если метки будут иметь фиксированную длину (как было показано в главе 8), однако сетки точнее отражают табличную структуру интерфейсов, включая формы ввода, а также большие таблицы в целом. Следующий раздел иллюстрирует эти различия в программном коде.

Сравнение методов grid и pack

Настало время сделать некоторые сравнения и противопоставления: пример 9.19 реализует одинаковые расцвеченные формы ввода с помощью методов `grid` и `pack`, чтобы легче было обнаружить различия между двумя подходами.

Пример 9.19. PP4E\Gui\Tour\Grid\grid2.py

```
"""
добавляет эквивалентное окно, используя фреймы-ряды и метки фиксированной длины;
использование фреймов-колонок не обеспечивает точного взаимного расположения
виджетов Label и Entry по горизонтали; программный код в обоих случаях имеет
одинаковую длину, хотя применение встроенной функции enumerate позволило бы
сэкономить 2 строки в реализации компоновки по сетке;
"""

from tkinter import *
colors = ['red', 'green', 'orange', 'white', 'yellow', 'blue']

def gridbox(parent):
    "компоновка по номерам рядов/колонок в сетке"
    row = 0
    for color in colors:
        lab = Label(parent, text=color, relief=RIDGE, width=25)
        ent = Entry(parent, bg=color, relief=SUNKEN, width=50)
        lab.grid(row=row, column=0)
        ent.grid(row=row, column=1)
        ent.insert(0, 'grid')
        row += 1

    def packbox(parent):
        "фреймы-ряды и метки фиксированной длины"
        for color in colors:
```

```
            row = Frame(parent)
            lab = Label(row, text=color, relief=RIDGE, width=25)
            ent = Entry(row, bg=color, relief=SUNKEN, width=50)
            row.pack(side=TOP)
            lab.pack(side=LEFT)
            ent.pack(side=RIGHT)
            ent.insert(0, 'pack')

if __name__ == '__main__':
    root = Tk()
    gridbox(Toplevel())
    packbox(Toplevel())
    Button(root, text='Quit', command=root.quit).pack()
    mainloop()
```

Версия на основе метода `pack` в этом примере использует фреймы-ряды и метки фиксированной длины (напомню еще раз, что схема с фреймами-колонками не позволяет добиться требуемой точности). Эти две функции создают виджеты меток и полей ввода одинаковым способом, но размещаются они совершенно разными путями:

- При использовании метода `pack` метки и поля ввода прикрепляются к левому и правому краям с помощью параметров `side`, и для каждого ряда создается виджет Frame (который прикрепляется к верхнему краю родителя).
- При использовании метода `grid` каждому виджету назначается положение с помощью параметров `row` (ряд) и `column` (колонка) в предполагаемой табличной сетке родителя.

Как мы уже знаем, при использовании метода `pack` порядок добавления виджетов имеет большое значение: виджет получает остаток стороны родительского контейнера, к которой он прикрепляется (что в данном случае неактуально), и элементы, прикрепляемые первыми, обрезаются последними (метки и самые верхние строки здесь обрезаются последними). При компоновке с помощью метода `grid` тот же эффект обрезания достигается благодаря особенностям поведения сетки. Если запустить этот сценарий, он создаст окна, изображенные на рис. 9.30, – по одному окну для каждой схемы размещения.

При внимательном рассмотрении можно заметить, что разница в объеме программного кода, необходимого для каждой схемы, ничтожна, по крайней мере для случая простой формы. Схема на основе метода `pack` должна создать отдельный виджет `Frame` для каждого ряда, а схема на основе метода `grid` должна отслеживать номер текущего ряда.

Фактически обе схемы требуют для реализации одинаковое количество строк программного кода. Однако справедливости ради следует заметить, что реализации обеих схем можно было бы сократить на одну строку, если добавлять метку немедленно, а реализацию на основе метода `grid` можно было бы сократить еще на две строки, применив встро-

енную функцию `enumerate`, чтобы избавиться от необходимости вести счет рядов вручную. Ниже приводится уменьшенная версия функции `gridbox`:

```
def gridbox(parent):
    for (row, color) in enumerate(colors):
        Label(parent,text=color,relief=RIDGE,width=25).grid(row=row,column=0)
        ent = Entry(parent, bg=color, relief=SUNKEN, width=50)
        ent.grid(row=row, column=1)
        ent.insert(0, 'grid')
```

Оставим дальнейшее уменьшение размеров реализации для тех, кому не чужд спортивный интерес (мы приводим не самую шокирующую реализацию, однако ваше стремление к краткости программного кода совсем не обязательно покажется достижением вашим коллегам!). Независимо от схемы компоновки сложность обеих реализаций выглядит примерно одинаковой. Однако, как будет показано далее, для реализации компоновки по сетке может потребоваться больше программного кода, когда появляется необходимость изменять размеры виджетов вместе с окном.

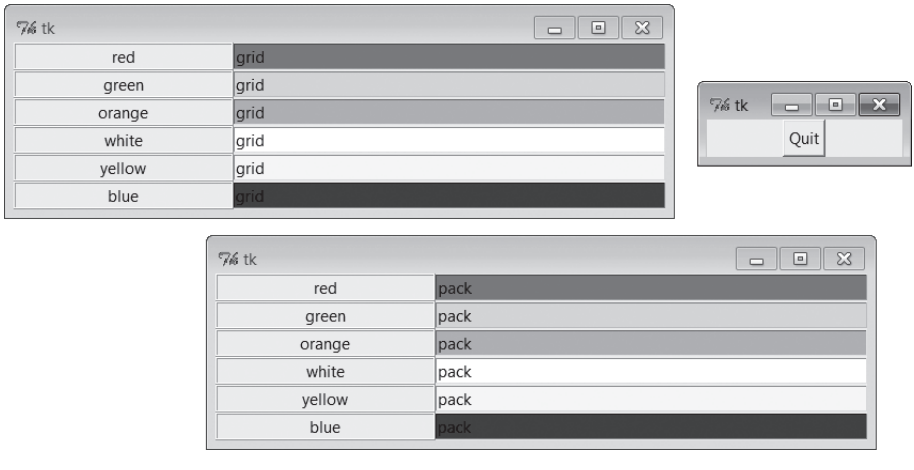


***Рис. 9.30.** Эквивалентные окна для схем размещения на основе grid и pack*

Сочетание grid и pack

Обратите внимание, что в примере 9.19 каждой функции-конструктору формы передается совершенно новый виджет `Toplevel`, благодаря чему версии на основе методов `grid` и `pack` создают различные окна верхнего уровня. Так как два менеджера компоновки не могут одновременно использоваться в одном родительском окне, необходимо следить за тем, чтобы по недосмотру не смешать их. Пример 9.20 демонстрирует возможность компоновки виджетов с помощью методов `pack` и `grid` в одном

и том же окне, но только после заключения их в отдельные контейнерные виджеты Frame.

Пример 9.20. PP4E\Gui\Tour\Grid\grid2-same.py

```
"""
создает формы с применением методов pack и grid в отдельных фреймах в одном
и том же окне; методы grid и pack не могут одновременно использоваться в одном
родительском контейнере (например, в корневом окне), но могут использоваться
в разных фреймах в одном и том же окне;
"""

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()
```

Если запустить этот сценарий, получится составное окно с двумя формами идентичного вида (рис. 9.31), но эти два вложенные фрейма в действительности управляются совершенно разными менеджерами компоновки.

С другой стороны, такой программный код, как приводится в примере 9.21, вызывает грубую ошибку, поскольку пытается вызывать методы pack и grid в одном и том же родителе – только один менеджер компоновки может использоваться в каждом отдельном родительском окне.

Пример 9.21. PP4E\Gui\Tour\Grid\grid2-fails.py

```
"""
ОШИБКА -- методы pack и grid не могут одновременно использоваться в одном и том
же родительском контейнере (здесь, корневое окно)
"""

from tkinter import *
from grid2 import gridbox, packbox

root = Tk()
gridbox(root)
packbox(root)
```

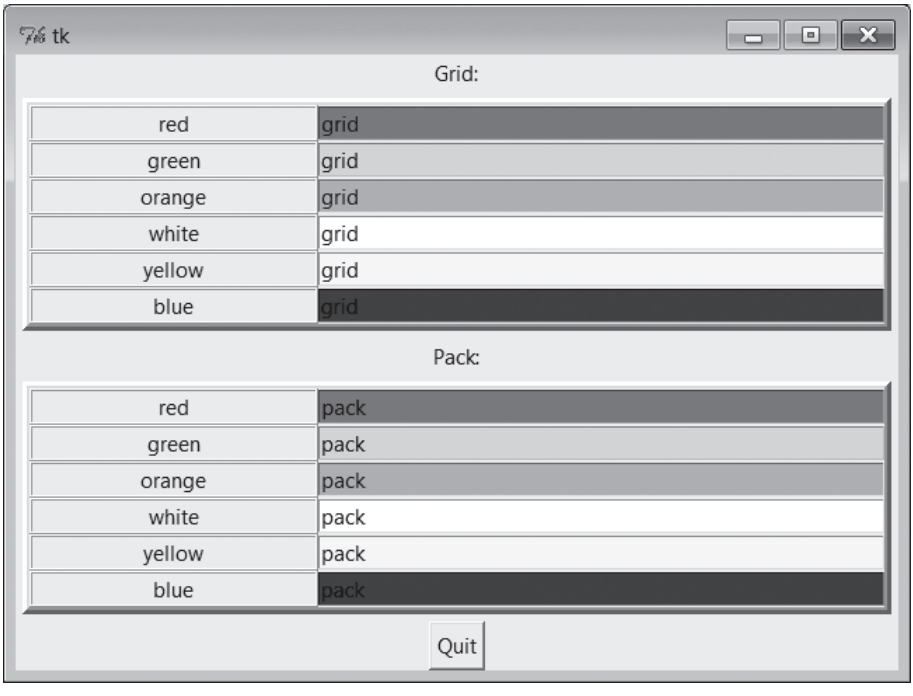


Рис. 9.31. *grid и pack в одном окне*

```
Button(root, text='Quit', command=root.quit).pack()
mainloop()
```

Этот сценарий передает каждой из функций одного и того же родителя (окно верхнего уровня), пытаясь вывести обе формы в одном окне. На моей машине он полностью подвешивает процесс Python, не выводя вообще никаких окон (в некоторых версиях Windows мне пришлось прибегнуть к Ctrl+Alt+Delete, чтобы уничтожить процесс, в других версиях достаточно было перезапустить программу Командная строка (Command Prompt)).

Комбинирование менеджеров компоновки может представлять сложность, пока с этим не освоишься. Чтобы сделать этот сценарий работоспособным, например, требуется просто изолировать форму с сеткой в собственном родительском контейнере, чтобы оградить его от влияния метода pack, используемого в корневом окне, – как демонстрирует следующая альтернативная реализация, где изменения выделены полужирным шрифтом:

```
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 напомню, что в настоящее время внутри одного родителя допускается использовать либо метод `pack`, либо метод `grid`, но не тот и другой одновременно. Возможно, в будущем это давнее ограничение будет снято, что, впрочем, маловероятно с учетом различий в схемах двух менеджеров компоновки, но на всякий случай проверьте свою версию Python.

Реализация возможности растягивания виджетов, размещаемых по сетке

А теперь некоторые практические замечания: сетки, которые мы видели до сих пор, имеют фиксированный размер – они не увеличиваются в размере при увеличении размеров содержащего их окна. Пример 9.22 реализует чрезвычайно патриотическую форму ввода с применением обоих методов, `grid` и `pack`, но в нем выполняются дополнительные настройки, необходимые, чтобы обеспечить растягивание всех виджетов в обоих окнах вместе со своими окнами.

Пример 9.22. PP4E\Gui\Tour\Grid\grid3.py

```
"добавляет метку в верхней части окна и возможность растягивания форм"

from tkinter import *
colors = ['red', 'white', 'blue']

def gridbox(root):
    Label(root, text='Grid').grid(columnspan=2)
    row = 1
    for color in colors:
        lab = Label(root, text=color, relief=RIDGE, width=25)
        ent = Entry(root, bg=color, relief=SUNKEN, width=50)
        lab.grid(row=row, column=0, sticky=NSEW)
        ent.grid(row=row, column=1, sticky=NSEW)
        root.rowconfigure(row, weight=1)
        row += 1
    root.columnconfigure(0, weight=1)
    root.columnconfigure(1, weight=1)

def packbox(root):
    Label(root, text='Pack').pack()
    for color in colors:
        row = Frame(root)
        lab = Label(row, text=color, relief=RIDGE, width=25)
        ent = Entry(row, bg=color, relief=SUNKEN, width=50)
        row.pack(side=TOP, expand=YES, fill=BOTH)
        lab.pack(side=LEFT, expand=YES, fill=BOTH)
        ent.pack(side=RIGHT, expand=YES, fill=BOTH)
```

```
root = Tk()
gridbox(Toplevel(root))
packbox(Toplevel(root))
Button(root, text='Quit', command=root.quit).pack()
mainloop()
```

Если запустить этот сценарий, он создаст картину, изображенную на рис. 9.32. Снова создаются отдельные окна для методов `pack` и `grid` 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 несколько более темно-серый).

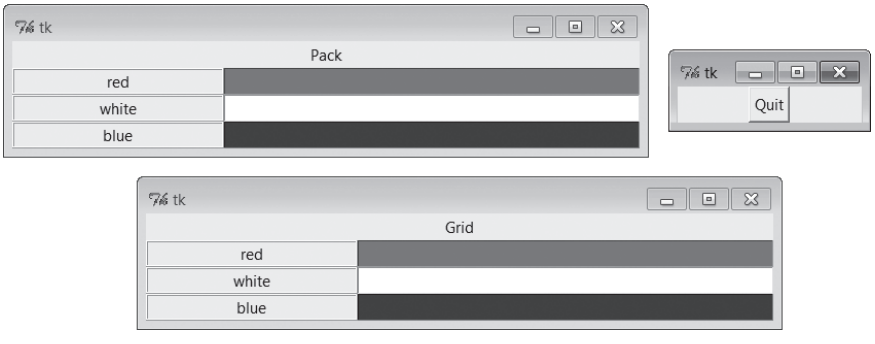


Рис. 9.32. *Окна для схем размещения на основе grid и pack до изменения размеров*

Однако на этот раз изменение размеров обоих окон с помощью мыши заставляет все встроенные в них метки и поля ввода растягиваться вместе с окнами, как показано на рис. 9.33 (где в поля ввода был введен текст).

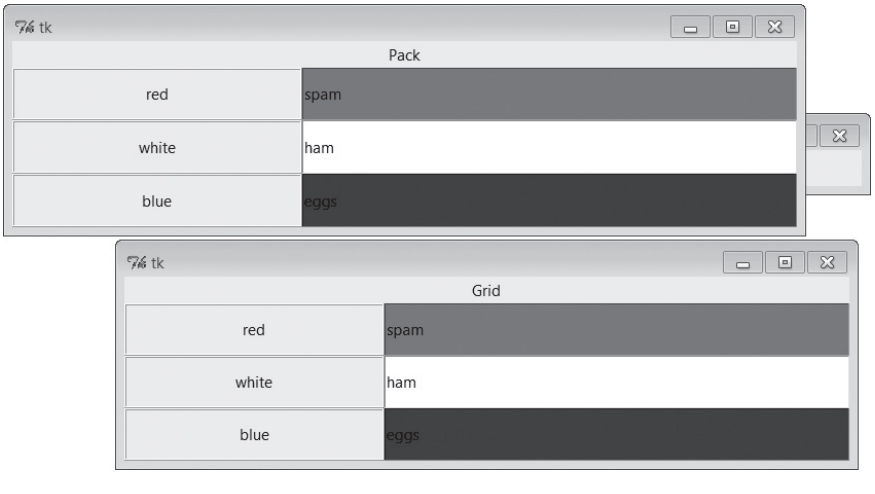


Рис. 9.33. *Окна для схем размещения на основе grid и pack после изменения размера*

В данной реализации при уменьшении размеров окна со схемой размещения на основе метода pack первыми обрезаются виджеты, которые были добавлены последними. В окне со схемой размещения на основе метода grid обрезаются все метки и поля ввода сразу, в отличие от сценария grid2, где демонстрируется поведение по умолчанию (попробуйте у себя на компьютере).

Изменение размеров в сетках

Теперь, когда я показал, что делают эти окна, нужно объяснить, как они это делают. В главе 7 мы узнали, как заставить графические элементы растягиваться при использовании метода pack: мы использовали параметры expand и fill, чтобы увеличить отводимое им пространство и заставить их растягиваться в пределах этого пространства соответственно. Чтобы обеспечить растягивание виджетов, размещаемых с помощью метода grid, требуется использовать другие протоколы. Ряды и колонки становятся растягиваемыми, когда они помечены с помощью параметра weight (вес), а виджеты растягиваются в отведенных им ячейках сетки, когда помечены с помощью параметра sticky (липкий):

Тяжелые ряды и колонки

При использовании метода pack ряды становятся растягиваемыми, если способность к растягиванию придается соответствующему виджету Frame, в результате задания значений параметров expand=YES и fill=BOTH. Для сетки нужно быть несколько конкретнее: чтобы обеспечить полную способность к растягиванию, требуется вызвать метод rowconfigure контейнера сетки для каждого ряда и метод columnconfigure для каждой колонки. Обоим методам нужно передать параметр weight веса со значением больше нуля, чтобы ряды и колонки стали растягиваемыми. По умолчанию вес принимается равным нулю (что означает отсутствие поддержки растягивания), а контейнером сетки в данном сценарии служит просто окно верхнего уровня. Использование разных весов для разных рядов и колонок заставляет их растягиваться в различных пропорциях.

Липкие виджеты

При использовании метода pack виджеты растягиваются по горизонтали или вертикали, заполняя отведенное им пространство, если передать этому методу параметр fill, а для позиционирования виджетов в отведенном им пространстве используется параметр anchor. Параметр sticky метода grid играет роли обоих параметров, fill и anchor, метода pack. Чтобы заставить растягиваться виджеты, размещаемые по сетке, можно прилепить их к одному краю отведенной им ячейки (как с помощью параметра anchor) или более чем к одному краю (как с помощью параметра fill). Приклеивать виджеты можно в четырех направлениях — N (север), S (юг), E (восток) и W (запад), а комбинируя эти четыре буквы, можно обеспечить приклеивание сразу к нескольким сторонам. Например, значение W в параметре

sticky обеспечит выравнивание виджета по левому краю отведенного ему пространства (подобно anchor=W в методе pack), а значение NS заставит виджет растягиваться по вертикали в выделенном пространстве (подобно fill=Y в методе pack).

Приклеивание виджетов не использовалось в предыдущих примерах, потому что структуру интерфейса образовывали виджеты с постоянными размерами (виджеты были не меньше пространства, отведенного им в ячейке сетки), а изменение размеров вообще не поддерживалось. В данном случае для параметра sticky определено в значение NSEW, чтобы графические элементы растягивались во всех направлениях вместе с отведенными им ячейками.

Различные сочетания весов рядов и колонок, а также значений параметра sticky создают различные эффекты при изменении размеров. Например, если удалить вызов метода columnconfigure из сценария grid3, это приведет к тому, что интерфейс будет растягиваться только в вертикальном направлении. Попробуйте сами поэкспериментировать с этими настройками и посмотреть, к каким эффектам это приведет.

Объединение колонок или рядов

Есть еще одно важное отличие в том, как сценарий grid3 настраивает свои окна. Оба окна – с методами grid и pack – выводят вверху метку, которая размещается по ширине всего окна. В схеме размещения на основе метода pack метка просто прикрепляется к верхнему краю окна в целом (напомню, что параметр 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 в строгом мире сеток размещение такой метки потребует приложить дополнительные усилия. В первой строке функции, реализующей схему размещения по сетке, это делается следующим образом:

```
Label(root, text='Grid').grid(columnspan=2)
```

Чтобы виджет охватывал сразу несколько колонок, методу grid передается параметр columnspan с указанием количества охватываемых колонок. В данном случае он указывает, что метка в верхней части 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 и преимуществом, и недостатком – в зависимости от того, насколько равномерно должны располагаться виджеты; эти два параметра установки диапазонов позволяют при необходимости организовать исключения из правила.

Так какой же менеджер компоновки оказывается здесь победителем? Если имеет значение изменение размеров, как в этом сценарии, то под-

ход на основе сетки оказывается несколько более сложным (в данном примере для реализации размещения по сетке потребовалось написать три дополнительных строки программного кода). С другой стороны, использование функции `enumerate` снова может изменить общий счет, метод `grid` остается удобным для создания простых форм, да и ваши схемы компоновки на основе методов `grid` и `pack` могут быть другими.

Дополнительная информация о способах компоновки элементов форм ввода приводится в разделе, где обсуждаются утилиты мастеров форм, которые мы реализуем ближе к концу главы 12 и будем использовать в главе 13, при разработке пользовательского интерфейса программы передачи файлов и клиента FTP. Как будет показано далее, автоматизировав процедуру создания привлекательных форм, мы сможем избавить себя от необходимости вникать в детали позднее. Кроме того, в главе 11 мы реализуем менее обычную компоновку формы в диалоге замены программы PyEdit и при размещении полей заголовков электронного письма в примере PyMailGUI, в главе 14.

Создание крупных таблиц с помощью grid

До сих пор мы строили наборы меток и полей ввода из двух колонок. Это типичный вид форм ввода, но менеджер `grid` в библиотеке tkinter способен организовывать значительно более крупные матрицы. Так, в примере 9.23 создается массив меток, состоящий из пяти строк и четырех колонок, в котором каждая метка просто выводит номер своей строки и колонки (`row.col`). Если запустить этот сценарий, он создаст окно, изображенное на рис. 9.34.

Пример 9.23. PP4E\Gui\Tour\Grid\grid4.py

```
# простая двухмерная таблица, в корневом окне Tk по умолчанию

from tkinter import *

for i in range(5):
    for j in range(4):
        lab = Label(text='%d.%d' % (i, j), relief=RIDGE)
        lab.grid(row=i, column=j, sticky=NSEW)

mainloop()
```

Если вы предположили, что это выглядит как способ программирования электронных таблиц, то вы, вероятно, на правильном пути. Пример 9.24 дает дальнейшее развитие этой мысли и добавляет кнопку, которая выводит в поток `stdout` текущие значения полей ввода в таблице (обычно в окно консоли).

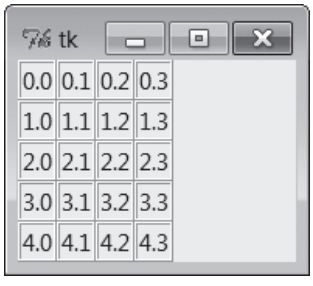


***Рис. 9.34.** Массив 5×4 меток с координатами*

Пример 9.24. PP4E\Gui\Tour\Grid\grid5.py

```
# двухмерная таблица полей ввода, корневое окно Tk по умолчанию

from tkinter import *

rows = []
for i in range(5):
    cols = []
    for j in range(4):
        ent = Entry(relief=RIDGE)
        ent.grid(row=i, column=j, sticky=NSEW)
        ent.insert(END, '%d.%d' % (i, j))
        cols.append(ent)
    rows.append(cols)

def onPress():
    for row in rows:
        for col in row:
            print(col.get(), end=' ')
        print()

Button(text='Fetch', command=onPress).grid()
mainloop()
```

Если запустить этот сценарий, он создаст окно, изображенное на рис. 9.35, и сохранит все виджеты полей ввода в сетке в двухмерном списке списков. При нажатии кнопки Fetch сценарий выполнит обход списка списков полей ввода, чтобы получить и отобразить все текущие значения в сетке. Ниже приводится вывод после двух нажатий кнопки Fetch – одного перед изменениями в полях ввода и другого после:

```
C:\...\PP4E\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
```

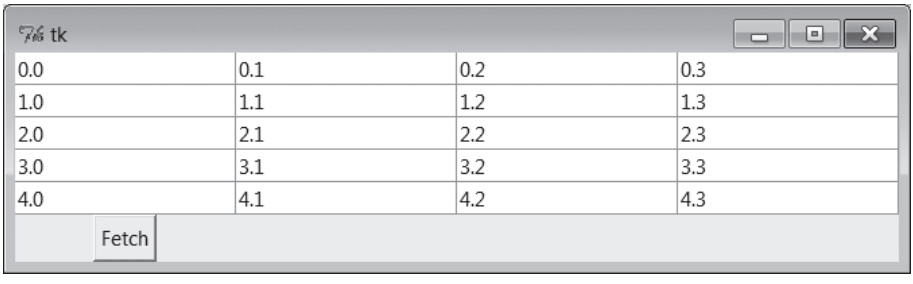


***Рис. 9.35.** Более крупная сетка полей ввода*

Теперь, когда мы знаем, как создавать и выполнять обход массивов полей ввода, добавим еще несколько полезных кнопок. В примере 9.25 добавлен еще один ряд, в котором отображаются суммы по столбцам, и кнопки, которые обнуляют все поля и вычисляют суммы по столбцам.

Пример 9.25. PP4E\Gui\Tour\Grid\grid5b.py

```
# добавляет суммирование по столбцам и очистку полей ввода

from tkinter import *
numrow, numcol = 5, 4

rows = []
for i in range(numrow):
    cols = []
    for j in range(numcol):
        ent = Entry(relief=RIDGE)
        ent.grid(row=i, column=j, sticky=NSEW)
        ent.insert(END, '%d.%d' % (i, j))
        cols.append(ent)
    rows.append(cols)

sums = []
for i in range(numcol):
    lab = Label(text='?', relief=SUNKEN)
    lab.grid(row=numrow, column=i, sticky=NSEW)
    sums.append(lab)

def onPrint():
    for row in rows:
```

```
            for col in row:
                print(col.get(), end=' ')
            print()
        print()

    def onSum():
        tots = [0] * numcol
        for i in range(numcol):
            for j in range(numrow):
                tots[i] += eval(rows[j][i].get())  # вычислить сумму
        for i in range(numcol):
            sums[i].config(text=str(tots[i]))      # отобразить в интерфейсе

    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. 9.36 изображено окно этого сценария после вычисления сумм по четырем столбцам чисел. Чтобы получить таблицу другого размера, измените переменные `numrow` и `numcol` в начале сценария.

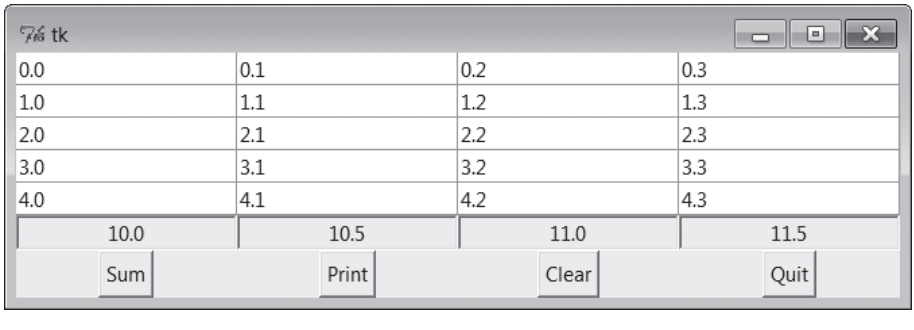


Рис. 9.36. *Добавление суммирования по столбцам*

И наконец, пример 9.26, реализующий еще одно, последнее расширение, которое оформлено как класс, что обеспечивает возможность повторного его использования. Кроме того, в нем добавлена кнопка загрузки таблицы из файла с данными. Предполагается, что файл содержит строку

для каждого ряда данных, а внутри строки колонки разделяются пробельными символами (пробелами или символами табуляции). При загрузке данных из файла автоматически изменяется размер таблицы, чтобы уместить все колонки.

Пример 9.26. PP4E\Gui\Tour\Grid\grid5c.py

```
#реализация в виде встраиваемого класса

from tkinter import *
from tkinter.filedialog import askopenfilename
from PP4E.Gui.Tour.quitter import Quitter # повт. использование, pack и grid

class SumGrid(Frame):
    def __init__(self, parent=None, numrow=5, numcol=5):
        Frame.__init__(self, parent)
        self.numrow = numrow              # я – контейнерный фрейм
        self.numcol = numcol              # компоновку выполняет вызвавшая пр.,
        self.makeWidgets(numrow, numcol)  # иначе можно было бы использовать
                                          # единственным способом
    def makeWidgets(self, numrow, numcol):
        self.rows = []
        for i in range(numrow):
            cols = []
            for j in range(numcol):
                ent = Entry(self, relief=RIDGE)
                ent.grid(row=i+1, column=j, sticky=NSEW)
                ent.insert(END, '%d.%d' % (i, j))
                cols.append(ent)
            self.rows.append(cols)
        self.sums = []
        for i in range(numcol):
            lab = Label(self, text='?', relief=SUNKEN)
            lab.grid(row=numrow+1, column=i, sticky=NSEW)
            self.sums.append(lab)

        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(), end=' ')
            print()
        print()

    def onSum(self):
        tots = [0] * self.numcol
        for i in range(self.numcol):
```

```
                for j in range(self.numrow):
                    tots[i] += eval(self.rows[j][i].get()) # суммировать данные
            for i in range(self.numcol):
                self.sums[i].config(text=str(tots[i]))

        def onClear(self):
            for row in self.rows:
                for col in row:
                    col.delete('0', END)                  # удалить содержимое
                    col.insert(END, '0.0')                # зарезерв. значение
            for sum in self.sums:
                sum.config(text='?')

        def onLoad(self):
            file = askopenfilename()
            if file:
                for row in self.rows:
                    for col in row: col.grid_forget()     # очистить интерфейс
                for sum in self.sums:
                    sum.grid_forget()

                filelines = open(file, 'r').readlines()   # загрузить данные
                self.numrow = len(filelines)              # изменить размер табл.
                self.numcol = len(filelines[0].split())
                self.makeWidgets(self.numrow, self.numcol)

                for (row, line) in enumerate(filelines):  # загрузить в интерфейс
                    fields = line.split()
                    for col in range(self.numcol):
                        self.rows[row][col].delete('0', END)
                        self.rows[row][col].insert(END, fields[col])

    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 не применяет к себе самому ни метод `grid`, ни метод `pack`. Чтобы дать возможность прикрепления к контейнерам, где есть другие графические элементы, скомпонованные тем или иным способом, он оставляет управление собственной компоновкой неопределенным и требует, чтобы вызывающая программа сама применяла метод `pack` или `grid` к его экземплярам. Контейнеры могут выбрать любую схему компоновки для своих дочерних элементов, потому что они независимы в своем выборе, но прикрепляе-

мым классам компонентов, предназначенным для использования с любыми менеджерами компоновки, нельзя поручить управлять собой, так как они не могут заранее знать политику своего родителя.

Это довольно длинный пример, в котором нет почти ничего нового в отношении компоновки по сетке или виджетов в целом, поэтому я оставлю его для самостоятельного изучения и просто покажу, что он делает. На рис. 9.37 изображено начальное окно, созданное этим сценарием, после того как была изменена последняя колонка и произведено суммирование – не забудьте включить корневой каталог *PP4E* дерева с примерами в путь поиска модулей (например, в переменную окружения PYTHONPATH), чтобы сценарий смог импортировать пакет.

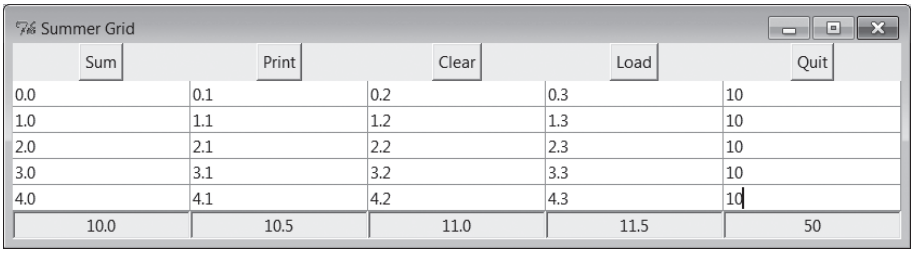


Рис. 9.37. *Добавлена загрузка данных из файла*

По умолчанию класс создает сетку размером 5 на 5, но существует возможность определять другие размерности, как в конструкторе класса, так и в командной строке сценария. При нажатии кнопки Load выводится стандартный диалог выбора файла, с которым мы встречались ранее (рис. 9.38).

Файл данных *grid5-data1.txt* содержит семь строк и шесть колонок данных:

```
C:\...\PP4E\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 графический интерфейс соответствующим образом изменяются размеры сетки – класс просто заново выполняет логику создания виджетов после удаления прежних элементов ввода с помощью метода grid_forget. Метод grid_forget отвязывает виджеты в сетке и в результате удаляет их с экрана. Другие способы удаления и перерисовки компонентов графического интерфейса предоставляют методы

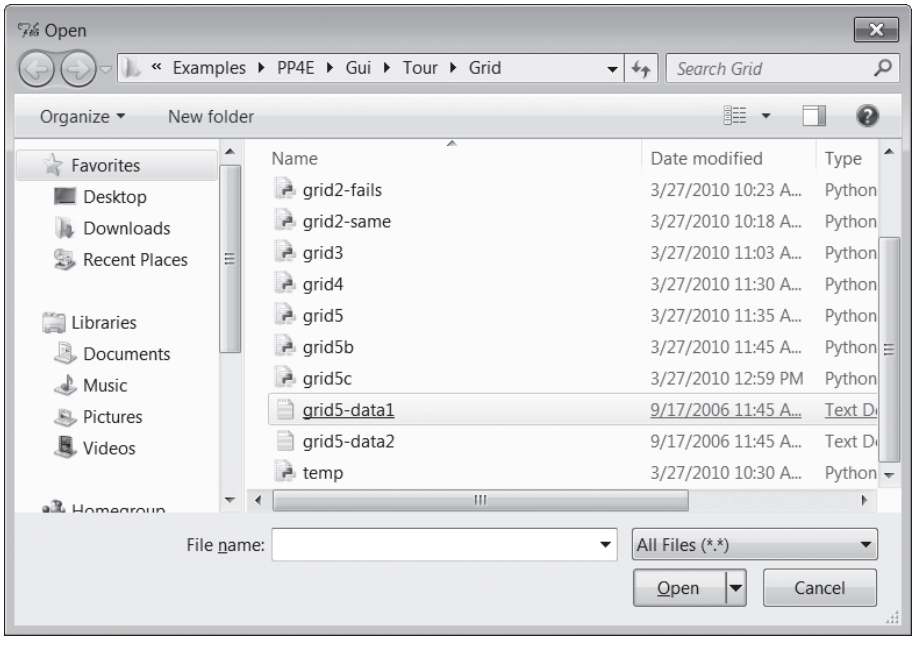


Рис. 9.38. *Диалог открытия файла в сценарии SumGrid*

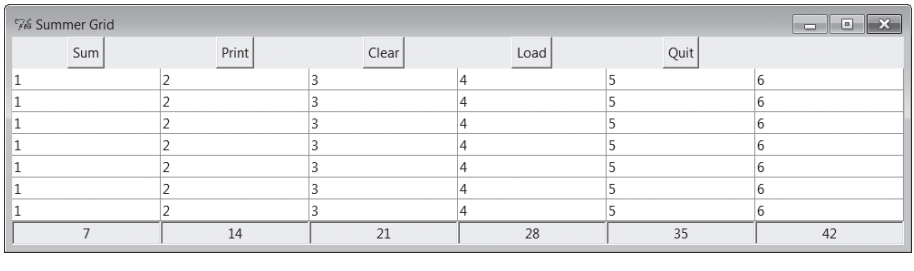


Рис. 9.39. *Файл с данными загружен, отображен и просуммирован*

`pack_forget` виджетов и `withdraw` окна, которые используются в обработчике события `after` примеров «будильников» в следующем разделе.

На рис. 9.39 показано, как выглядит окно после операций удаления и перерисовки виджетов, выполненных в результате щелчков на кнопках Load и Sum.

Файл с данными *grid5-data2.txt* имеет те же размерности, но в двух колонках он содержит не просто числа, а выражения. Так как этот сценарий преобразует значения полей ввода с помощью встроенной функции `eval`, в полях этой таблицы допускается использовать любые выраже-

ния Python, если они могут быть вычислены в области видимости метода onSum:

```
C:\...\PP4E\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 программный код на языке Python, что иллюстрирует рис. 9.40. Эта особенность может оказаться достаточно мощной. Представьте себе, например, полноценную сетку электронной таблицы – значения полей могут быть «фрагментами» программного кода на языке Python, которые динамически вычисляют значения, вызывают функции из модулей и даже загружают текущие котировки акций из Интернета с помощью инструментов, с которыми мы познакомимся в следующей части книги.

Однако эта особенность может представлять *опасность* – в поле может содержаться выражение, удаляющее содержимое вашего жесткого диска![1] Если вы не до конца уверены в том, какими могут быть полученные выражения, не используйте функцию eval (осуществляйте преобразование, применяя более ограниченные функции, такие как int и float) или обеспечьте выполнение процесса Python с ограниченными правами доступа к системным компонентам, которые было бы нежелательно подвергать опасности.

---

[1] Я решил обратить ваше внимание на это, поскольку считаю, что понимание опасности позволит в будущем избежать ее – если процесс Python обладает правом удаления файлов, передача функции eval строки с программным кодом __import__('os').system('rm -rf *') в Unix приведет к вызову команды оболочки, которая удалит все файлы в текущем каталоге и во вложенных в него подкаталогах (в Windows аналогичный эффект можно получить с помощью команды 'rmdir /S /Q .'). Не делайте этого! Чтобы увидеть менее опасное и более полезное применение этой особенности, введите выражение __import__('math').pi в одну из ячеек таблицы – щелчок на кнопке Sum вычислит значение pi (3.14159). Также безопасно будет передать функции eval выражение "__import__('os').system('dir')" в интерактивном сеансе. Все вышесказанное относится и к встроенной функции exec – функция eval выполняет строки выражений, функция exec – инструкции, а выражения являются инструкциями (но не наоборот). Разумеется, обычный пользователь графического интерфейса едва ли введет подобный программный код случайно, особенно если этот пользователь вы сами, но будьте внимательны!

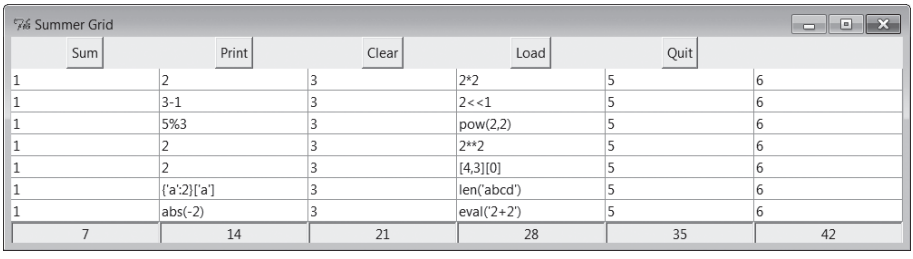


Рис. 9.40. *Выражения на языке Python в данных и таблице*

Конечно, этому сценарию еще очень далеко до настоящей электронной таблицы. Сценарий подсчитывает суммы по колонкам и способен загружать данные из файлов, но ячейки не могут содержать формулы, ссылающиеся на другие ячейки. Однако из-за недостатка места в книге дальнейшие улучшения для достижения этой цели я оставляю читателям в качестве упражнения.

Я должен также отметить, что о размещении по сетке можно сказать больше, чем позволяет объем книги. Например, путем создания вложенных фреймов с собственными сетками можно строить более сложные структуры в виде иерархий компонентов, во многом подобно тому, как размещает вложенные фреймы менеджер компоновки `pack`. А теперь перейдем к последней теме обзора виджетов.

Инструменты синхронизации, потоки выполнения и анимация

Последняя остановка в нашей экскурсии по виджетам, вероятно, самая необычная. Библиотека tkinter предоставляет ряд инструментов, которые связаны с моделью программирования, управляемого событиями, а не с отображением графики на экране.

Некоторым приложениям с графическим интерфейсом требуется периодически выполнять действия в фоновом режиме. Например, чтобы придать виджету «мерцающий» вид, можно зарегистрировать обработчик, который будет вызываться через равные промежутки времени. Аналогично при выполнении длительной операции с файлом неверно было бы заблокировать прочие действия в графическом интерфейсе – если бы удалось заставить цикл событий периодически выполнять обновления, графический интерфейс мог бы оставаться активным. В библиотеке tkinter есть средства для планирования таких отложенных действий и принудительного обновления интерфейса:

`widget.after(`*`milliseconds`*`, `*`function`*`, *`args`*`)`

Этот инструмент планирует вызов указанной функции по истечении заданного числа миллисекунд. Данная форма вызова не останавли-

вает программу – функция обратного вызова будет запущена позднее из обычного цикла событий tkinter, а вызывающая программа продолжит свою работу как обычно и графический интерфейс останется активным, пока функция ожидает вызова. Как уже говорилось в главе 5, в отличие от объекта `Timer` из модуля `threading`, события `widget.after` распространяются в главном потоке выполнения графического интерфейса и потому могут выполнять в нем любые изменения.

Аргумент *`function`* может быть любым вызываемым объектом Python: функцией, связанным методом, `lambda`-выражением и так далее. Аргумент *`milliseconds`* определяет интервал времени в миллисекундах и является целым числом – если разделить значение этого аргумента на 1000, получится эквивалентное число секунд. Любые значения в кортеже `args` будут переданы функции *`function`* в виде позиционных аргументов.

На практике вместо отдельных аргументов можно использовать `lambda`-выражения, чтобы сделать связь аргументов с функцией более явной, но это не является обязательным. Когда в качестве функции передается связанный метод, он может получать дополнительную информацию из атрибутов объекта, а не из аргументов. Метод `after` возвращает идентификатор, который можно передать методу `after_cancel`, чтобы отменить вызов обработчика. Метод `after` используется очень часто, поэтому несколько ниже о нем будет рассказываться более подробно и с примерами.

`widget.after(`*`milliseconds`*`)`

Этот инструмент останавливает выполнение программы на заданное количество миллисекунд. Например, если передать в аргументе число 5000, программа будет приостановлена на 5 секунд. В сущности, это то же самое, что библиотечная функция Python `time.sleep(`*`seconds`*`)`, и обе функции могут применяться для создания задержки при отображении (например, в анимационных программах, таких как PyDraw и более простых примерах ниже).

`widget.after_idle(`*`function`*`, *`*`args`*`)`

Этот инструмент планирует вызов указанной функции при отсутствии других событий, которые должны обрабатываться. То есть функция *`function`* становится обработчиком холостого времени, который вызывается, когда графический интерфейс не занят ничем другим.

`widget.after_cancel(`*`id`*`)`

Этот инструмент отменяет вызов обработчика, запланированный методом `after` до того, как он произойдет. Аргумент *`id`* – значение, возвращаемое методом `after`.

`widget.update()`

Этот инструмент вынуждает библиотеку tkinter обработать все ожидающие события, имеющиеся в очереди событий, в том числе изменение геометрии окна, а также обновление и перерисовку виджетов. Его можно периодически вызывать из долго выполняющегося обработчика, 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. На время работы обработчика, выполняющегося продолжительное время, интерфейс 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 перекрытии или открытии другими окнами, пока не произойдет возврат из обработчика.

Например, программы, осуществляющие анимацию путем последовательного перемещения объекта и приостановки, должны вызывать этот метод, не дожидаясь конца анимации, иначе на экране можно будет увидеть только конечное положение объекта. Что еще хуже, интерфейс окажется совершенно неактивным, пока не произойдет возврат из обработчика анимации (смотрите простые примеры воспроизведения анимации далее в этой главе и в программе PyDraw в главе 11).

`widget.update_idletasks()`

Этот инструмент запускает обработку всех событий холостого времени. Иногда он безопаснее, чем метод `after`, который в некоторых случаях может стать причиной возникновения состояния гонки за ресурсами (race conditions). События холостого времени используются виджетами Tk для отображения самих себя.

`_tkinter.createfilehandler(`*`file`*`, `*`mask`*`, `*`function`*`)`

Этот инструмент назначает функцию, которая будет вызываться при изменении состояния файла. Функция может быть вызвана, когда в файле появятся данные для чтения, когда он станет доступным для записи или когда будет возбуждено исключение. В аргументе *`file`* передается объект Python файла или сокета (формально – любой объект с методом `fileno()`) или целочисленный дескриптор файла; аргументе *`mask`* – значение `tkinter.READABLE` или `tkinter.WRITABLE`, определяющее режим; а в аргументе *`function`* передается функция обратного вызова, принимающая два аргумента – признак готовности файла к выполнению операции и маску. Обработчики файлов часто используются для обработки каналов и сокетов, так как обычные функции ввода/вывода могут блокировать вызывающую программу.

Этот метод недоступен в Windows и потому не будет рассматриваться в данной книге. Поскольку он доступен только в Unix, для разработ-

ки переносимых графических интерфейсов лучше использовать циклы таймера `after` для проверки готовности к выполнению операции и порождать потоки выполнения, которые будут читать данные и помещать их в очередь при необходимости; более подробно этот прием описывается в главе 10. Потоки выполнения являются более универсальным механизмом выполнения неблокирующих операций передачи данных.

`widget.wait_variable(`*`var`*`)`

`widget.wait_window(`*`win`*`)`

`widget.wait_visibility(`*`win`*`)`

Эти инструменты приостанавливают выполнение вызвавшей программы до момента, когда переменная tkinter изменит свое значение, будет разрушено окно или окно станет видимым. Все они входят в локальный цикл событий, благодаря чему функция `mainloop` приложения продолжает обработку событий. Обратите внимание, что аргумент *`var`* является объектом переменной tkinter (о которых рассказывалось ранее), а не простой переменной Python. Для использования в модальных диалогах сначала следует вызвать `widget.focus()` (чтобы установить фокус ввода) и `widget.grab()` (чтобы сделать окно единственным активным).

Некоторые из этих инструментов мы будем использовать в примерах, но не станем вникать во все их особенности здесь. За дополнительной информацией обращайтесь к другой документации по библиотекам Tk и tkinter.

Использование потоков выполнения в графических интерфейсах tkinter

Имейте в виду, что во многих программах поддержка потоков выполнения в Python, с которой мы познакомились в главе 5, способна отчасти решать те же задачи, что и инструменты tkinter, перечисленные в предыдущем разделе, и даже позволяет использовать их. Например, чтобы избежать блокировки интерфейса (и не заставлять пользователей бездействовать) во время продолжительных операций обмена данными через файлы или сокеты, этот обмен можно выполнять в дочерних потоках выполнения, при этом остальная часть программы будет выполняться, как обычно. Аналогично графические интерфейсы, ожидающие появления данных в канале или в сокете, могут использовать для этих целей потоки выполнения, обработчики, устанавливаемые методом `after`, или их комбинации, и тем самым избежать блокирования графического интерфейса.

Однако при использовании потоков выполнения в программах на основе библиотеки tkinter обращения к графическому интерфейсу должны выполняться только из главного потока (в котором был создан интерфейс и запущена функция `mainloop`). По крайней мере, потоки не

должны пытаться одновременно изменять графический интерфейс. Например, метод `update`, описанный в предыдущем разделе, исторически является источником проблем в многопоточных графических интерфейсах – если вызывать его (или другой метод, вызывающий `update`) в порожденных потоках выполнения, он иногда может приводить к неожиданному и даже эффектному краху программы.

Чтобы увидеть простые и яркие примеры, демонстрирующие небезопасность обращения к графическим интерфейсам на базе tkinter из нескольких потоков выполнения, загляните в следующие сценарии, находящиеся в дереве примеров к книге, и попробуйте запустить их:

...\PP4E\Gui\Tour\threads-demoAll-frm.py
...\PP4E\Gui\Tour threads-demoAll-win.py

Эти сценарии являются версиями примеров 8.32 и 8.33 из предыдущей главы, которые конструируют четыре демонстрационных компонента графического интерфейса в параллельно выполняющихся потоках. Оба они зависают в Windows и требуют принудительного завершения. Но, хотя некоторые операции с графическим интерфейсом могут безопасно выполняться параллельно в разных потоках (например, смотрите пример 9.32, где выполняется перемещение элементов на холсте), тем не менее в целом библиотека tkinter не поддерживает многопоточную модель выполнения. (Дополнительные доказательства этого утверждения вы найдете в обсуждении многопоточной реализации циклов обновления в следующей главе, сразу после примера 10.28 – в этом примере поток выполнения пытается вывести новое окно, что вызывает сбой в работе графического интерфейса.)

Такое отношение к потокам выполнения со стороны реализации поддержки графических интерфейсов может измениться со временем, но на сегодняшний день оно налагает некоторые структурные ограничения. Например, порожденные потоки выполнения обычно не могут производить опеарции с графическим интерфейсом, поэтому, в случае необходимости, они должны взаимодействовать с главным потоком программы, используя глобальные переменные или разделяемые объекты, такие как очереди. Например, поток, ожидающий получения данных из сокета, может добавлять их в разделяемые очереди или просто устанавливать глобальные переменные и инициировать изменения в графическом интерфейсе через обработчик, устанавливаемый методом `after`. А обработчик может обрабатывать результаты, полученные в порождаемых потоках.

Некоторые операции над графическим интерфейсом поддерживают возможность выполнения в многопоточном режиме, тем не менее программы с графическим интерфейсом лучше делить на главный поток, управляющий графическим интерфейсом, и ряд «рабочих» потоков, не связанных с интерфейсом, что поможет избежать потенциальных конфликтов и решить проблему поддержки многопоточности в целом. Про-

грамма PyMailGUI, представленная далее в книге, например, вызывает функции-обработчики, сохраняемые потоками выполнения в очереди.

Не забывайте также, что независимо от наличия поддержки многопоточной модели выполнения в графических интерфейсах многопоточные программы с графическим интерфейсом должны следовать тем же правилам, что и любые другие многопоточные программы. Как мы узнали в главе 5, такие программы всегда должны синхронизировать доступ к совместно используемым данным, если есть вероятность, что сразу несколько потоков попытаются изменить их. Модель организации потоков производитель/потребитель, основанная на очередях, способна снять множество проблем, тем не менее в программах, порождающих потоки выполнения, изменяющие информацию, которую использует главный поток управления графическим интерфейсом, по-прежнему может требоваться использовать блокировки, чтобы избежать проблем, связанных с попытками одновременного изменения совместно используемых данных.

Подробнее многопоточные графические интерфейсы мы будем рассматривать в главе 10, а в четвертой части книги познакомимся с более реалистичными примерами многопоточных графических интерфейсов, таких как PyMailGUI в главе 14. В PyMailGUI, например, чтобы избежать блокирования интерфейса, для выполнения продолжительных операций используются потоки, все действия с графическим интерфейсом производятся только в главном потоке, а для предотвращения конфликтов, возможных при изменении совместно используемых данных, применяются блокировки.

Использование метода after

Из всех инструментов, перечисленных выше, наиболее интересным является метод `after`. Он позволяет сценариям назначить обработчик, который будет вызван в некоторый момент времени в будущем. Несмотря на его простоту, он будет часто использоваться в последующих примерах. В частности, в главе 11 мы познакомимся с программой часов, которая с помощью метода `after` просыпается 10 раз в секунду и получает текущее время, а также с программой показа слайдов, которая с помощью метода `after` устанавливает интервал смены фотографий (программы PyClock и PyView). Для иллюстрации основ планирования вызова обработчиков служит пример 9.27.

Пример 9.27. PP4E\Gui\Tour\alarm.py

```
# мигает и издает сигнал каждую секунду, используя цикл с методом after()

from tkinter import *

class Alarm(Frame):
    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()

    def repeater(self):                          # каждые N миллисекунд
        self.bell()                              # подать сигнал
        self.stopper.flash()                     # мигнуть кнопкой
        self.after(self.msecs, self.repeater)   # запланировать следующий вызов

if __name__ == '__main__': Alarm(msecs=1000).mainloop()
```

Этот сценарий создает окно, изображенное на рис. 9.41, и периодически вызывает метод `flash` кнопки, заставляющий кнопку мигнуть (изменяет ее цвет на короткое время), и метод `bell`, который обращается к функции подачи звукового сигнала. Метод `repeater` вызывает методы `beep` и `flash` и с помощью метода `after` устанавливает обработчик, который будет выполнен через определенный промежуток времени.

Рис. 9.41. *Прекратите пищать!*

Метод `after` не останавливает вызывающий сценарий: обработчик вызывается в фоновом режиме, в то время как программа выполняет другую работу, – технически с того момента, когда цикл событий Tk получит возможность обнаружить изменение времени. Для этого метод `repeater` каждый раз вызывает `after` и заново устанавливает обработчик. Отложенные события являются одноразовыми: чтобы повторить событие, его нужно снова запланировать.

В итоге этот сценарий начинает подавать звуковые сигналы и мигать кнопкой, как только будет выведено его окно с одной кнопкой, и продолжает подавать сигналы и мигать снова и снова. Прочие действия и операции с графическим интерфейсом не влияют на это. Даже если свернуть окно, сигналы будут продолжаться, потому что события таймера tkinter генерируются в фоновом режиме. Чтобы прекратить сигналы, нужно закрыть окно или щелкнуть на кнопке. Изменив задержку `msecs`, можно заставить сигнал звучать так часто или так редко, как позволяет система (максимально допустимая частота может зависеть от платформы). Предупреждаю заранее, что это не лучшая демонстрационная программа для запуска в многолюдном помещении.

Скрытие и перерисовка виджетов и окон

Метод flash кнопки вызывает кратковременное изменение цвета виджета, но с помощью метода config так же просто можно динамически изменять и другие параметры внешнего вида виджетов, таких как кнопки, метки и текст,. Например, эффекта мигания можно добиться путем инвертирования цветов переднего и заднего плана элементов вручную, вызывая метод config в обработчиках, установленных методом after. Ради забавы в примере 9.28 приводится версия сценария, подающего звуковой сигнал, в которой сделан еще один шаг.

Пример 9.28. PP4E\Gui\Tour\alarm-hide.py

```
# стирает и отображает кнопку в обработчике, устанавливаемом методом after()

from tkinter import *
import alarm

class Alarm(alarm.Alarm):                       # измените обработчик таймера
    def __init__(self, msecs=1000):             # по умолчанию = 1 секунда
        self.shown = False
        alarm.Alarm.__init__(self, msecs)

    def repeater(self):                         # каждые N миллисекунд
        self.bell()                             # подать сигнал
        if self.shown:
            self.stopper.pack_forget()          # скрыть кнопку
        else:                                   # или изменить цвет, мигнуть...
            self.stopper.pack()
        self.shown = not self.shown             # изменить до следующего раза
        self.after(self.msecs, self.repeater)  # переустановить обработчик

if __name__ == '__main__': Alarm(msecs=500).mainloop()
```

Если запустить этот сценарий, на экране появится то же самое окно, но теперь при каждом событии от таймера кнопка поочередно будет стираться и отображаться вновь. Метод pack_forget виджета стирает нарисованный элемент, а метод pack отображает его снова – методы grid_forget и grid аналогичным образом скрывают и отображают виджеты в сетке. Метод pack_forget удобно использовать для динамического изменения графического интерфейса. Например, можно решить, какие компоненты должны отображаться в тот или иной момент времени, создавать виджеты заранее и отображать их только по мере надобности. В данном случае это просто значит, что пользователь должен щелкнуть на кнопке, пока она видна, иначе шум будет продолжаться.

Сценарий в примере 9.29 идет еще дальше. Здесь с помощью нескольких методов реализовано скрытие и появления всего окна:

- Чтобы скрыть и отобразить не какой-то отдельный виджет, а целое окно, можно воспользоваться методами withdraw и deiconify этого окна. Метод withdraw, используемый в примере 9.29, полностью сти-

рает окно и его ярлык (если необходимо, чтобы ярлык окна оставался видимым, используйте метод `iconify`).

- Метод `lift` поднимает окно над всеми другими окнами или над определенным окном, переданным методу в виде аргумента. Этот метод также может также вызываться под именем `tkraise`, но не `raise` – его именем в Tk, потому что `raise` в языке Python является зарезервированным словом.
- Метод `state` возвращает или изменяет текущее состояние окна – он принимает значения `normal`, `iconic`, `zoomed` (на весь экран) и `withdrawn`.

Поэкспериментируйте с этими методами, чтобы понять, чем они отличаются. Их также можно использовать для динамического вывода предварительно созданных диалогов, однако практическая ценность этого приема невелика.

Пример 9.29. PP4E\Gui\Tour\alarm-withdraw.py

```
# то же самое, но скрывает и отображает окно целиком

from tkinter import *
import alarm

class Alarm(alarm.Alarm):
    def repeater(self):                          # каждые N миллисекунд
        self.bell()                              # подать сигнал
        if self.master.state() == 'normal':      # окно отображается?
            self.master.withdraw()               # скрыть окно, без ярлыка
        else:                                    # iconify свертывает в ярлык
            self.master.deiconify()              # иначе перерисовать окно
            self.master.lift()                   # и поднять над остальными
        self.after(self.msecs, self.repeater)   # переустановить обработчик

if __name__ == '__main__': Alarm().mainloop()  # master = корневое окно Tk
                                               # по умолчанию
```

Этот сценарий действует точно так же, за исключением того, что при подаче сигнала появляется или исчезает все окно – закрывать его надо тогда, когда оно видно. Реализацию обработчика, вызываемого по таймеру, можно разнообразить массой других эффектов. Будете ли вы добиваться, чтобы ваши кнопки и окна мигали и исчезали, зависит скорее от мнения пользователей, чем от возможностей библиотеки tkinter.

Простые приемы воспроизведения анимации

Все графические интерфейсы, представленные до сих пор в этой книге, за исключением примера `canvasDraw` с непосредственным перемещением фигур, были довольно статичными. В данном, последнем разделе будет показано, как можно изменить эту ситуацию, добавив в пример 9.16 несколько простых анимаций перемещения фигуры на холсте.

Здесь также демонстрируется и расширяется понятие тегов холста – операции перемещения в этом примере применяются сразу ко всем объектам на холсте, связанным с тегом. Все овалы перемещаются при нажатии клавиши 0, а все прямоугольники – при нажатии клавиши R. Как уже отмечалось ранее, методы холста принимают не только идентификаторы объектов, но и имена тегов.

Но главная задача сейчас состоит в том, чтобы проиллюстрировать простые приемы анимации с помощью инструментов, основанных на измерении интервалов времени и описанных выше в этом разделе. Существует три основных способа перемещения объектов по холсту:

- С помощью циклов, использующих функцию `time.sleep` для приостановки на доли секунды между последовательными операциями перемещения, наряду с вызовами метода `update` вручную. Сценарий выполняет перемещение, приостанавливается, передвигает объект еще немного и так далее. Функция `time.sleep` приостанавливает работу вызывающей программы и не возвращает управление в цикл событий графического интерфейса – обработка операций с интерфейсом, выполняемых во время перемещения, откладывается. Из-за этого после каждого перемещения нужно вызывать метод `canvas.update`, чтобы перерисовать экран, иначе экран не обновится, пока не закончится весь цикл перемещения в обработчике и не произойдет возврат. Это классический пример обработчика, выполняющегося продолжительное время. Без вызова метода обновления экрана вручную никакие другие события графического интерфейса не будут обработаны до возврата из обработчика (даже перерисовка окна).
- С помощью метода `widget.after`, планирующего выполнение операций перемещения через каждые несколько миллисекунд. Поскольку этот подход основан на расписании событий, которые библиотека tkinter отправляет обработчикам, он допускает параллельное осуществление нескольких перемещений и не требует вызова метода `canvas.update`. Для выполнения перемещений используется цикл событий, поэтому приостановка программы не требуется и графический интерфейс не блокируется.
- С помощью потоков выполнения, в которых выполняется несколько экземпляров циклов с приостановкой вызовом метода `time.sleep`, как в первом подходе. Так как потоки выполняются параллельно, приостановка любого из потоков не блокирует ни графический интерфейс, ни другие потоки, выполняющие перемещения. Как уже описывалось выше, графический нтерфейс вообще не следует обновлять из порожденных потоков, но некоторые методы холста, в частности метод `move`, в настоящее время допускают возможность вызова из потоков выполнения.

Из этих трех схем первая обеспечивает самое плавное воспроизведение анимации, но она замедляет другие операции во время перемеще-

ния. Вторая схема дает более замедленное перемещение, чем остальные, но в целом безопаснее, чем использование потоков выполнения, и обе последние схемы позволяют одновременно передвигать несколько объектов.

Использование циклов time.sleep

В следующих трех разделах поочередно демонстрируется структура программного кода для всех трех подходов, создающая новые подклассы примера canvasDraw, с которым мы познакомились в примере 9.16. Обращайтесь к этому примеру за информацией о привязке других событий и об основах выполнения операций рисования, перемещения и стирания. Здесь объекты, создаваемые на холсте, ассоциируются с тегами, а также добавляются новые операции и выполняется привязка новых событий. Пример 9.30 иллюстрирует первый подход.

Пример 9.30. PP4E\Gui\Tour\canvasDraw_tags.py

```
"""
перемещение с применением тегов и функции time.sleep (без помощи метода widget.
after или потоков выполнения); функция time.sleep не блокирует цикл событий
графического интерфейса на время паузы, но интерфейс не обновляется до выхода из
обработчика или вызова метода widget.update; текущему вызову обработчика onMove
уделяется исключительное внимание, пока он не вернет управление: если в процессе
перемещения нажать клавишу 'R' или 'O';
"""

from tkinter import *
import canvasDraw, time

class CanvasEventsDemo(canvasDraw.CanvasEventsDemo):
    def __init__(self, parent=None):
        canvasDraw.CanvasEventsDemo.__init__(self, parent)
        self.canvas.create_text(100, 10, 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 овалы с данным тегом
```

```
    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()    # принудительно обновить изображение
                time.sleep(0.25)        # пауза, не блокирующая интерфейс

if __name__ == '__main__':
    CanvasEventsDemo()
    mainloop()
```

Все три сценария в этом разделе при вытягивании новых фигур с помощью левой кнопки мыши создают окно с голубыми овалами и красными прямоугольниками. Сама реализация вытягивания наследуется из суперкласса. Щелчок правой кнопкой мыши немедленно перемещает одну фигуру, а двойной щелчок левой кнопкой по-прежнему очищает холст – эти операции также унаследованы из суперкласса. В действительности в этом новом сценарии лишь изменены методы, создающие объекты, – теперь они ассоциируют создаваемые объекты с тегами и окрашивают их в соответствующие цвета, добавлено текстовое поле в верхней части холста и добавлены обработчики событий, выполняющие перемещение. На рис. 9.42 показано, как выглядит окно этого подкласса после создания нескольких фигур.

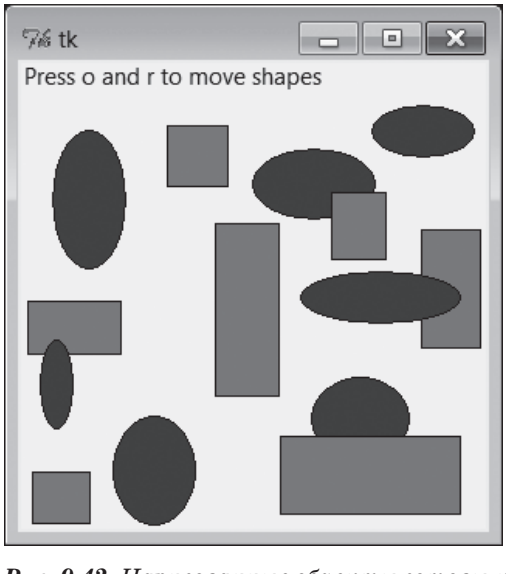


***Рис. 9.42.** Нарисованные объекты готовы к анимации*

С помощью клавиш O и R начинается анимация всех нарисованных овалов и прямоугольников соответственно. Например, при нажатии клавиши O начинают синхронно перемещаться все голубые овалы. Объекты, которые подвергаются анимации, помечаются пятью квадратами вокруг своего местоположения, и перемещаются со скоростью четыре шага в секунду. Новые объекты, которые вытягиваются, когда другие находятся в движении, тоже начинают перемещаться, потому что помечены тегами. Вам обязательно следует запустить этот сценарий, чтобы получить представление о простой анимации, которую он реализует (можно, конечно, попробовать подвигать влево-вправо и вверх-вниз книгу, но это все-таки не совсем то, что нужно, да и может глупо выглядеть на людях).

Использование событий widget.after

Главный недостаток первого подхода в том, что одновременно может происходить только одна анимация: если нажать клавишу R или O во время движения, новый запрос приостанавливает предыдущее перемещение до своего окончания, потому что каждый обработчик операции перемещения допускает только один поток управления при своей работе. То есть в каждый конкретный момент времени может выполняться только один цикл, использующий `time.sleep`, а новый вызов этой функции из метода `update` фактически является рекурсивным вызовом, который приостанавливает уже выполняющийся цикл.

Обновление экрана во время перемещений тоже происходит несколько замедленно, потому что производится, только когда метод `update` вызывается вручную (попробуйте вытянуть фигуру или перекрыть окно другим окном во время перемещения и вы в этом убедитесь сами). Фактически если закомментировать вызов метода `update` в примере 9.30, графический интерфейс вообще перестанет откликаться во время выполнения операций перемещения – он не будет перерисовываться при перекрытии другими окнами, не будет откликаться на действия пользователя и никакого эффекта анимации воспроизводиться не будет (по истечении времени вы просто увидите окно в заключительном состоянии). Это полноценная имитация влияния операций, выполняющихся продолжительное время, на графический интерфейс.

Пример 9.31 переопределяет метод `moveInSquares`, чтобы снять такие ограничения, – применяя метод `after`, он обеспечивает перемещение практически без пауз. Кроме того, он демонстрирует наиболее часто используемый (и, вероятно, лучший) способ обработки событий от таймера в графических интерфейсах на основе библиотеки tkinter. Разбиение задания на части вместо того чтобы выполнять его целиком, позволяет выполнить естественное распределение частей по времени и выполнять несколько заданий одновременно.

Пример 9.31. PP4E\Gui\Tour\canvasDraw_tags_after.py

```
"""
аналогично, но с применением метода widget.after() вместо циклов time.sleep;
поскольку это планируемые события, появляется возможность перемещать овалы
и прямоугольники _одновременно_ и отпадает необходимость вызывать метод update
для обновления графического интерфейса; движение станет беспорядочным, если еще
раз нажать 'o' или 'r' в процессе воспроизведения анимации: одновременно начнут
выполняться несколько операций перемещения;
"""

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()
```

Эта версия наследует все изменения из предыдущей версии и при этом позволяет перемещать овалы и прямоугольники одновременно – нарисуйте несколько овалов и прямоугольников, а затем нажмите клавишу `O` и затем сразу 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, начнет перемещаться немедленно.

Использование нескольких потоков выполнения с циклами time.sleep

Иногда того же эффекта можно добиться выполнением анимации в потоках. Как уже говорилось выше, в целом обновлять интерфейс из порожденного потока выполнения опасно, но в данном примере этот прием действует (по крайней мере, на платформах, участвовавших в тестировании). В примере 9.32 каждая задача анимации выполняется как независимый и параллельный поток. Это означает, что при каждом нажатии клавиши `O` или `R` для запуска анимации порождается новый поток, который выполняет эту задачу.

Пример 9.32. PP4E\Gui\Tour\canvasDraw_tags_thread.py

```
"""
аналогично, но анимация воспроизводится с применением циклов time.sleep,
выполняемых параллельно в разных потоках, а не с помощью обработчиков
событий, устанавливаемых методом after(), или одного активного цикла time.
sleep; поскольку потоки выполняются параллельно, эта версия также позволяет
перемещать овалы и прямоугольники _одновременно_ и не требует вызывать метод
update для обновления графического интерфейса: фактически вызов метода .update()
в этой версии приводит к краху, хотя некоторые методы холста можно безопасно
использовать 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)       # приостанавливает только этот поток

    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 9.31, но на этот раз оно выполняется с помощью параллельных 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 периоды выполнения обработчиков перемещений могут перекрываться во времени – функция `time.sleep` блокирует только вызвавший ее поток, а не программу в целом.

В настоящее время этот пример прекрасно работает в Windows, но в Linux он однажды потерпел у меня неудачу – интерфейс не обновлялся при изменении его в потоках, и никаких изменений не наблюдалось до появления последующих событий. Правило, которое гласит, что желательно избегать изменения графического интерфейса в порождаемых потоках выполнения, остается верным. Обычно надежнее использовать потоки только для вычислений, а какое-либо обновление экрана производить в главном потоке (создавшем графический интерфейс). Тем не менее даже в этой модели главный поток выполнения может следить за результатами работы других потоков с помощью метода `after`, как

в примере 9.31, не приостанавливаясь в периоды ожидания (подробнее об этом рассказывается в следующем разделе и в следующей главе).

Не исключено, что реализация составных частей, участвующих в создании анимации, изменится со временем, и не исключено, что возможность обновления графического интерфейса из потоков выполнения будет лучше поддерживаться в следующих версиях tkinter, поэтому ищите дополнительные сведения об изменениях в поддержке многопоточной модели в новых выпусках.

Другие темы, связанные с анимацией

Мы снова обратимся к анимации в примере PyDraw в главе 11. В нем будут возрождены все три приема – задержки, таймеры и потоки – для перемещения фигур, текста и фотографий в произвольные точки холста, помечаемые щелчком мыши. И хотя система абсолютных координат холста – главная рабочая лошадка для реализации большинства нетривиальных анимаций, в целом возможности анимации, основанные на библиотеке tkinter, ограничены лишь вашим воображением. В заключение скажу еще несколько слов, чтобы обозначить некоторые из имеющихся возможностей.

Другие анимационные эффекты

Помимо анимационных эффектов, которые создаются с применением холста, различные анимационные эффекты можно также создавать с помощью инструментов настройки виджетов. Как было показано ранее в примерах сценариев `alarm` (пример 9.28), где использовались эффекты скрытия и мигания виджетов, с помощью метода `after` можно легко динамически изменять внешний вид и других виджетов. С помощью циклов на основе таймера можно организовать мигание виджетов, динамически стирать и перерисовывать виджеты и окна, инвертировать или изменять цвет виджетов и так далее. Еще один пример из этой категории, где динамически изменяется шрифт и цвет (хотя эргономика этого примера вызывает большое сомнение), приводится во врезке «На досуге...», в главе 1.

Потоки выполнения и анимация

Приемы выполнения продолжительных операций в параллельных потоках приобретают особое значение, когда анимация должна оставаться активной, пока приложение выполняет какую-либо работу. Например, представьте себе программу, которая загружает большой объем данных из сети, производит тяжелые математические вычисления или выполняет другие продолжительные операции. Если графический интерфейс такой программы должен воспроизводить анимацию или както иначе отображать ход выполнения операции, периодически изменяя внешний вид виджета или перемещая объекты на холсте, – просто используйте метод `after`, как было показано выше, для периодического

вызова обработчика, который будет изменять графический интерфейс. В обработчике, установленном с помощью метода `after`, можно, к примеру, обновлять индикатор хода выполнения или счетчик.

Кроме того, саму продолжительную операцию, вероятно, лучше выполнять в параллельном потоке, чтобы графический интерфейс оставался активным, и воспроизводить анимацию, ожидая завершения операции. В противном случае графический интерфейс будет оставаться неактивным, пока операция не завершится и не вернет управление. Внутри обработчика, установленного вызовом метода `after`, главный поток выполнения, управляющий графическим интерфейсом, мог бы проверять переменные или объекты, изменяемые потоком, выполняющим продолжительную операцию, чтобы определить момент ее завершения.

В частности, когда в приложении одновременно выполняется более одной продолжительной операции, порожденные потоки выполнения могут также взаимодействовать с главным потоком графического интерфейса, сохраняя информацию в объекте Python `Queue`, которая будет обрабатываться реализацией графического интерфейса внутри обработчика, устанавливаемого методом `after`. В общем случае в очередь `Queue` допускается помещать даже объекты функций, которые могут вызываться реализацией графического интерфейса для его обновления.

В главе 10 мы еще раз вернемся к обсуждению приемов реализации многопоточных графических интерфейсов и будем использовать их в примере PyMailGUI, далее в этой книге. А пока имейте в виду, что организация вычислений в отдельных потоках выполнения позволяет графическому интерфейсу оставаться активным и воспроизводить анимацию или реагировать на действия пользователя, ожидая окончания вычислений.

Инструменты отображения графики и реализации игровых программ

Если только вы не прекратили играть в видеоигры сразу после появления игры Pong[1], вы наверняка понимаете, что приемы перемещения и воспроизведения анимации, продемонстрированные в этой главе, могут использоваться для реализации игровых программ, но только самых простых. Для реализации игровых программ с более высокими требованиями в Python имеются дополнительные инструменты поддержки графики и игр, которые мы не рассматривали здесь.

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. Другие сторонние инструменты, такие как OpenGL, Blender, PyGame, Maya

[1] Игра-симулятор пинг-понга, появившаяся в 1972 году и ставшая прародительницей таких известных игр, как Breakout и Arkanoid. – *Прим. перев.*

и VPython, обеспечивают еще более высокоуровневые средства отображения графики и анимации. Кроме того, система PyOpenGL обеспечивает поддержку Tk для построения графических интерфейсов. Ссылки на эти и другие инструменты ищите на веб-сайтах PyPI или в поисковых системах.

Если вас интересует разработка игровых программ, обратите внимание на PyGame и другие пакеты поддержки разработки игр на языке Python, а также обращайтесь к другим книгам и веб-ресурсам, посвященным этой тематике. Язык Python нечасто 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 трехмерной графики его роль может быть расширена еще больше. Ссылки на другие имеющиеся расширения для этой области можно найти на сайте *http://www.python.org*.

Конец экскурсии

На этом мы завершаем наш обзор библиотеки tkinter. Вы познакомились со всеми основными виджетами и инструменты, предварительный обзор которых был сделан в конце главы 7 (вернитесь туда, чтобы просмотреть общее описание области, рассмотренной в этом путешествии). Дополнительные сведения вы получите, когда все представленные здесь инструменты вновь появятся в более крупных примерах в главах 10 и 11 и в оставшейся части книги в целом. В некотором смысле последние несколько глав заложили основу для перехода к более крупным программам, следующим далее.

Другие виджеты и их параметры

Однако следует заметить, что наш тур не был исчерпывающим. Мы познакомились со всеми основными виджетами в арсенале tkinter и попутно овладели основами построения графических интерфейсов, тем не менее мы пропустили ряд более новых и более совершенных виджетов, появившихся в библиотеке tkinter недавно:

---

[1] Самая известная, пожалуй, компания *Eve Online* по производству игровых программ использует Python для создания сценариев и значительной доли функциональности – не только для серверной, но и для клиентской части. Она использует версию Stackless Python, чтобы обеспечить высокую отзывчивость интерфейса при большом количестве параллельно выполняющихся заданий. Из других известных компаний, использующих Python, можно назвать производителя игры Civilization IV и ныне несуществующую Origin Systems (в последних сообщениях говорилось, что их игра Ultima Online II должна была использовать Python для поддержки анимации).

`Spinbox`

Поле ввода `Entry` для выбора значения из множества или из диапазона

`LabelFrame`

Фрейм с заголовком и рамкой вокруг группы элементов

`PanedWindow`

Виджет менеджера компоновки, который может содержать множество виджетов, изменяющих свои размеры при перемещении линий-разделителей мышью

Кроме того, мы даже не упомянули ни об одном из виджетов в популярных расширениях библиотеки tkinter, таких как Pmw, Tix и ttk (описываются в главе 7), и не коснулись ни одного стороннего пакета. Например:

- Расширения Tix и ttk реализуют дополнительные параметры виджетов, обозначенные в главе 7, которые теперь входят в состав стандартной библиотеки Python.
- Перечень сторонних пакетов имеет тенденцию изменяться со временем, тем не менее уже сейчас они предоставляют виджеты деревьев, средства отображения разметки HTML, диалоги выбора шрифта, таблицы и многое другое, а также огромный пакет виджетов Pmw.
- Многие программы с графическим интерфейсом на основе библиотеки tkinter, такие как стандартная среда IDLE, включают диалоги выбора шрифта, виджеты деревьев и многое другое, что с успехом может использоваться вами в ваших приложениях.

Поскольку такие расширения пока еще для нас слишком сложны, чтобы их можно было охватить с пользой для дела, в интересах экономии места в книге мы оставим их освещение за другими ресурсами. В поисках более богатых возможностей для своих графических интерфейсов обязательно ознакомьтесь с описанием дополнительных виджетов в документации по tkinter, Tk, Tix, ttk и Pmw, посетите веб-сайт PyPI по адресу *http://python.org/* или поищите сторонние расширения для tkinter в Интернете.

Следует также заметить, что виджеты также обладают дополнительными параметрами настройки, о которых не упоминалось в этом обзоре. Ищите описания этих параметров в ресурсах по библиотекам Tk и tkinter. В библиотеке tkinter имеются и другие инструменты, аналогичные представленным здесь, тем не менее пространство в книге, которое я могу отвести для их освещения, ограничено, во-первых, моим издателем, а во-вторых – небесконечностью древесных ресурсов.

10

Приемы программирования графических интерфейсов

«Создание улучшенной мышеловки»

В этой главе мы продолжаем изучать создание графических интерфейсов пользователей с помощью Python и стандартной библиотеки tkinter, путем представления коллекции более сложных шаблонов и приемов программирования графических интерфейсов. В трех предшествующих главах мы познакомились с основами использования библиотеки tkinter. Здесь мы применим полученные знания для конструирования структур более высокого уровня, которые пригодятся в создании более крупных программ. То есть здесь мы перейдем к написанию собственного программного кода, реализующего программный слой над и вне базового набора инструментов tkinter, который с пользой будет применен в более практичных примерах далее в книге.

В этой главе мы рассмотрим следующие приемы:

- Реализация типичных операций с графическим интерфейсом в виде подмешиваемых классов
- Конструирование меню и панелей инструментов из шаблонных структур данных
- Добавление графических интерфейсов к инструментам командной строки
- Перенаправление потоков ввода-вывода в виджеты
- Динамическая переустановка обработчиков графического интерфейса
- Обертывание и автоматизация интерфейсов окон верхнего уровня

- Применение потоков выполнения и очередей для устранения блокирования графических интерфейсов
- Создание всплывающих окон в программах, не имеющих графического интерфейса
- Добавление графических интерфейсов в виде отдельных программ, подключаемых через сокеты и каналы

Как и другие главы этой книги, данная глава преследует двойную цель – не только изучение программирования графических интерфейсов, но и изучение более общих понятий программирования на языке Python, таких как объектно-ориентированное программирование (ООП) и повторное использование программного кода. Как мы увидим далее, создавая инструменты для работы с графическими интерфейсами на языке Python, мы упрощаем их применение в самых разных контекстах и программах.

Для связки со следующей главой эта глава завершается знакомством с панелями запуска PyDemos и PyGadgets – графических интерфейсов, используемых для запуска демонстрационных примеров. В книге приводится лишь малая часть этих программ, тем не менее мы рассмотрим их структуру достаточно подробно, чтобы вы могли самостоятельно исследовать их, отыскав в пакете с примерами.

Два предварительных замечания: во-первых, обязательно читайте программный код в листингах, выясняя подробности, отсутствующие в описании. Во-вторых, несмотря на небольшой объем примеров в этой главе, они демонстрируют приемы, которые найдут практическое применение в более реалистичных программах. Мы будем использовать эти приемы в более крупных примерах в следующей главе и на протяжении всей оставшейся части книги. Фактически разработанные здесь модули мы часто будем повторно использовать как инструменты в других программах из этой книги – программное обеспечение многократного использования должно использоваться снова и снова. Но для начала давайте приложим максимум усилий и создадим некоторые инструменты.

GuiMixin: универсальные подмешиваемые классы

Если вы читали предыдущие три главы, то наверняка заметили, что программный код, конструирующий нетривиальный графический интерфейс, может быть очень длинным, если каждый виджет создавать вручную. Приходится не только вручную связывать все виджеты, но нужно помнить десятки параметров, которые должны быть установлены. При такой стратегии программирование графических интерфейсов часто становится упражнением по вводу с клавиатуры и и выполнению операций копирования и вставки в текстовом редакторе.

Функции создания виджетов

Вместо того чтобы выполнять все операции вручную, правильнее было бы создать оболочку или как-то иначе максимально автоматизировать процесс построения графического интерфейса. Одним из решений является создание функций, обеспечивающих создание виджетов с типичными настройками и автоматизирующих процесс конструирования. Например, можно было бы определить функцию создания кнопки, реализующую все тонкости настройки и поддерживающую большинство необходимых нам кнопок. В примере 10.1 демонстрируется группа таких функций, создающих виджеты.

Пример 10.1. PP4E\Gui\Tools\widgets.py

```
"""
################################################################################
функции-обертки, упрощающие создание виджетов и опирающиеся на некоторые
допущения (например, режим растягивания); используйте словарь **extras
именованных аргументов для передачи таких параметров настройки, как ширина,
шрифт/цвет и других, и повторно компонуйте возвращаемые виджеты, если компоновка
по умолчанию вас не устраивает;
################################################################################
"""

from tkinter import *

def frame(root, side=TOP, **extras):
    widget = Frame(root)
    widget.pack(side=side, expand=YES, fill=BOTH)
    if extras: widget.config(**extras)
    return widget

def label(root, side, text, **extras):
    widget = Label(root, text=text, relief=RIDGE) # настройки по умолчанию
    widget.pack(side=side, expand=YES, fill=BOTH) # компонуется автоматически
    if extras: widget.config(**extras)            # применить все
    return widget                                 # дополнительные параметры

def button(root, side, text, command, **extras):
    widget = Button(root, text=text, command=command)
    widget.pack(side=side, expand=YES, fill=BOTH)
    if extras: 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: widget.config(**extras)
    return widget
```

```
if __name__ == '__main__':
    app = Tk()
    frm = frame(app, TOP) # программного кода теперь требуется намного меньше!
    label(frm, LEFT, 'SPAM')
    button(frm, BOTTOM, 'Press', lambda: print('Pushed'))
    mainloop()
```

Этот модуль опирается на некоторые допущения, касающиеся его использования клиентами, и обеспечивает автоматизацию типичных последовательностей операций конструирования виджетов, такие как размещение методом `pack`. В результате применение этого модуля позволяет уменьшить объем программного кода в импортирующих его программах. Если запустить модуль из примера 10.1 как самостоятельный сценарий, он создаст простое окно с меткой в выступающей рамке слева и с кнопкой справа, в случае щелчка на которой в поток `stdout` выводится сообщение. Оба виджета растягиваются вместе с окном. Запустите этот пример у себя – его окно действительно не содержит ничего нового для нас, а его программный код организован скорее как библиотека, чем сценарий, который позднее будет повторно использоваться в программе PyCalc, в главе 19.

Такое решение, основанное на функциях, может сократить объем необходимого программного кода. Однако функции не обеспечивают возможность специализации, как это позволяют классы при объектно-ориентированном подходе. Кроме того, они не являются методами и не обладают доступом к информации о состоянии объекта, представляющего элемент графического интерфейса.

Вспомогательные подмешиваемые классы

Другим способом может быть реализация общих методов в классе и наследование их при необходимости. Такие классы обычно называют *подмешиваемыми* (*mixin*), потому что их методы «подмешиваются» в другие классы. Подмешиваемые классы служат своего рода пакетами полезных во многих случаях инструментов, оформленных в виде методов. Эта идея близка к импортированию модулей, однако подмешиваемые классы могут обращаться к конкретному экземпляру, `self`, используя состояние конкретного объекта и унаследованные методы. Сценарий в примере 10.2 демонстрирует, как это делается.

Пример 10.2. PP4E\Gui\Tools\guimixin.py

```
"""
################################################################################
класс, "подмешиваемый" во фреймы: реализует общие методы вызова стандартных
диалогов, запуска программ, простых инструментов отображения текста и так далее;
метод quit требует, чтобы этот класс подмешивался к классу Frame (или его
производным)
################################################################################
"""
```

```
from tkinter import *
from tkinter.messagebox import *
from tkinter.filedialog import *
from PP4E.Gui.Tour.scrolledtext import ScrolledText # или tkinter.scrolledtext
from PP4E.launchmodes import PortableLauncher, System # или используйте модуль
                                                      # multiprocessing

class GuiMixin:
    def infobox(self, title, text, *args): # используются 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)       # вернет True или False

    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:
            Frame.quit(self)               # нерекурсивный вызов quit!

    def help(self):                                # переопределите более
        self.infobox('RTFM', 'See figure 1...') # подходящим

    def selectOpenFile(self, file="", dir="."): # испол-ся стандартные диалоги
        return askopenfilename(initialdir=dir, initialfile=file)

    def selectSaveFile(self, file="", dir="."):
        return asksaveasfilename(initialfile=file, initialdir=dir)

    def clone(self, args=()):    # необязательные аргументы конструктора
        new = Toplevel()         # создать новую версию
        myclass = self.__class__ # объект класса экземпляра (самого низшего)
        myclass(new, *args)      # прикрепить экземпляр к новому окну

    def spawn(self, pycmdline, wait=False):
        if not wait:                                  # запустить новый процесс
            PortableLauncher(pycmdline, pycmdline)() # запустить программу
        else:
            System(pycmdline, pycmdline)()           # ждать ее завершения

    def browser(self, filename):
        new = Toplevel()                          # создать новое окно
        view = ScrolledText(new, file=filename) # Text с полосой прокрутки
        view.text.config(height=30, width=85)   # настроить Text во фрейме
        view.text.config(font=('courier', 10, 'normal')) # моноширинный шрифт
```

```
            new.title("Text Viewer")                # атрибуты менеджера окон
            new.iconname("browser")                 # текст из файла будет
                                                    # вставлен автоматически
        """
        def browser(self, filename): # на случай, если импортирован
            new = Toplevel()         # модуль tkinter.scrolledtext
            text = ScrolledText(new, height=30, width=85)
            text.config(font=('courier', 10, 'normal'))
            text.pack(expand=YES, fill=BOTH)
            new.title("Text Viewer")
            new.iconname("browser")
            text.insert('0.0', open(filename, 'r').read() )
        """

    if __name__ == '__main__':
        class TestMixin(GuiMixin, Frame):  # автономный тест
            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)
                Button(self, text='spawn', command=self.other).pack(fill=X)
            def other(self):
                self.spawn('guimixin.py') # запустить себя в отдельном процессе

        TestMixin().mainloop()
```

Хотя пример 10.2 и ориентирован на графические интерфейсы, кроме этого он иллюстрирует архитектурные идеи. Класс `GuiMixin` реализует обычные операции со стандартными интерфейсами, которые не подвержены влиянию возможных изменений в реализации. На деле реализации некоторых методов этого класса все-таки изменились – при переходе от первого ко второму изданию этой книги вызовы функций из устаревшего модуля `Dialog` были заменены вызовами новых стандартных диалогов Tk; в четвертом издании изменился компонент для просмотра содержимого текстовых файлов и он теперь использует другой класс текстового виджета с прокруткой. Так как интерфейс класса в примере скрывает подобные детали, отпадает необходимость изменять использующие его программы, чтобы сделать доступными в них новые приемы.

В данном виде класс `GuiMixin` предоставляет методы для вызова стандартных диалогов, клонирования окон, запуска программ, просмотра текстовых файлов и так далее. Позднее, если обнаружится, что одни и те же методы приходится писать снова и снова, их можно будет добавить в такой подмешиваемый класс, и они немедленно станут доступны везде, где импортируется и внедряется этот класс. Более того, методы класса `GuiMixin` можно наследовать и использовать в существующем

виде либо переопределять в подклассах. Таковы естественные преимущества классов перед функциями.

Здесь есть несколько тонкостей, которые следует отметить особо:

- 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 обеспечивают более мощный способ упаковки многократно используемых компонентов, чем отдельные классы. При правильном применении подмешиваемый класс может дать значительно больше, чем обработчик единственной кнопки.
- Метод `clone` создает новый экземпляр самого нижнего в иерархии класса, который подмешивает класс `GuiMixin`, в новом окне верхнего уровня (`self.__class__` – это объект класса, из которого был создан экземпляр). Предполагается, что конструктор класса не требует никаких других аргументов, кроме ссылки на родительский контейнер. Он открывает новый независимый экземпляр окна (и передает конструктору любые дополнительные аргументы).
- Метод `browser` открывает в новом окне объект `ScrolledText`, который мы создали в главе 9, и заполняет его текстом из файла, который нужно просмотреть. Как отмечалось в предыдущей главе, существует также стандартный виджет `ScrolledText`, находящийся в модуле `tkinter.scrolledtext`, но он имеет иной интерфейс, не загружает содержимое файла автоматически и, возможно, будет объявлен устаревшим (хотя этого не происходит уже многие годы). Для справки в класс включена реализация метода, использующая этот виджет.
- Метод `spawn` запускает программу на языке Python в новом процессе и либо ждет его завершения, либо нет (в зависимости от аргумента `wait`, со значением по умолчанию `False` – обычно графический интерфейс не должен ждать завершения дочерней программы). Этот метод прост потому, что тонкости запуска скрыты в модуле `launchmodes`, представленном в конце главы 5. Класс `GuiMixin` способствует применению и сам применяет на практике приемы повторного использования программного кода.

Назначение класса `GuiMixin` состоит в том, чтобы служить библиотекой многократно используемых инструментальных методов, и как самостоятельный класс он, в сущности, бесполезен. В действительности для использования его нужно подмешивать в классы, наследующие класс `Frame`: метод `quit` предполагает, что он смешивается с классом `Frame`, а метод `clone` предполагает, что он смешивается с классом виджета. Чтобы удовлетворить этим ограничениям, находящаяся в конце реализация самотестирования объединяет класс `GuiMixin` с виджетом `Frame`.

На рис. 10.1 изображена картина, которая возникает при самотестировании после щелчка на кнопках clone и spawn, а затем на кнопке help в одной из трех копий окна. Поскольку щелчок на кнопке spawn запускает отдельный процесс, окно, созданное таким способом, остается на экране после закрытия всех остальных окон, а его закрытие не оказывает влияния на другие окна. Окно, созданное щелчком на кнопке clone, напротив, закрывается при закрытии главного окна, однако щелчок на кнопке X в копии окна закрывает только это окно. Не забудьте включить путь к каталогу *PP4E* в переменную окружения PYTHONPATH, чтобы обеспечить возможность импортирования пакетов в этом и в последующих примерах.

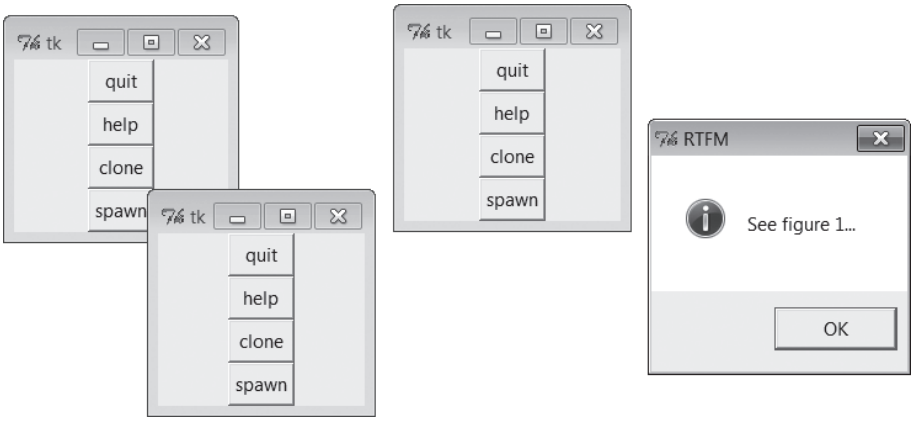


Рис. 10.1. *Реализация самотестирования класса GuiMixin в действии*

Мы снова встретимся с классом GuiMixin в роли подмешиваемого класса в последующих примерах – в конце концов, в этом весь смысл повторного использования кода. Хотя функции часто бывают полезными, тем не менее поддержка наследования классами, возможность доступа к информации в экземпляре и обеспечение дополнительной организационной структуры оказываются особенно полезными при создании графических интерфейсов. Например, если многие методы класса GuiMixin можно было бы заменить простыми функциями, то методы clone и quit – нет. В следующем разделе рассматриваются еще более широкие возможности подмешиваемых классов.

GuiMaker: автоматизация создания меню и панелей инструментов

Подмешиваемый класс из предыдущего раздела упрощает выполнение стандартных задач, но не решает проблем сложности связывания в таких виджетах, как меню и панели инструментов. Конечно, при на-

личии инструмента структурирования графического интерфейса, который генерировал бы программный код на языке Python, проблем бы не было. Мы бы проектировали виджеты интерактивно, нажимали кнопку и добавляли бы реализацию обработчиков.

Однако при использовании такого относительно простого инструмента, как tkinter, сгодится и подход, основанный на программировании. Хотелось бы иметь возможность, имея в окне шаблон для меню и панелей инструментов, наследовать некоторый класс, который сам выполнял бы всю черновую работу по конструированию. Ниже демонстрируется один из возможных способов, использующий деревья простых объектов. Класс в примере 10.3 интерпретирует структуры данных, содержащих представление меню и панелей инструментов, и автоматически создает необходимые виджеты.

Пример 10.3. PP4E\Gui\Tools\guimaker.py

```
"""
################################################################################
Расширенный Frame, автоматически создающий меню и панели инструментов в окне.
GuiMakerFrameMenu предназначен для встраивания компонентов (создает меню на
основе фреймов).
GuiMakerWindowMenu предназначен для окон верхнего уровня (создает меню Tk8.0).
Пример древовидной структуры приводится в реализации самотестирования (и
в PyEdit).
################################################################################
"""

import sys
from tkinter import *          # классы виджетов
from tkinter.messagebox import showinfo

class GuiMaker(Frame):
    menuBar = []               # значения по умолчанию
    toolBar = []               # изменять при создании подклассов
    helpButton = True          # устанавливать в start()

    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH) # растягиваемый фрейм
        self.start()           # в подклассе: установить меню/панель инстр.
        self.makeMenuBar()     # здесь: создать полосу меню
        self.makeToolBar()     # здесь: создать панель инструментов
        self.makeWidgets()     # в подклассе: добавить середину

    def makeMenuBar(self):
        """
        создает полосу меню вверху (реализация меню Tk8.0 приводится ниже)
        expand=no, fill=x, чтобы ширина оставалась постоянной
        """
        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:              # сканировать список вложенных элем.
            if item == 'separator':     # строка: добавить разделитель
                menu.add_separator({})
            elif type(item) == list:    # список: неактивных элементов
                for num in item:
                    menu.entryconfig(num, state=DISABLED)
            elif type(item[2]) != list:
                menu.add_command(label     = item[0], # команда: метка
                                 underline = item[1], # горячая клавиша
                                 command   = item[2]) # обр-к: вызыв. объект
            else:
                pullover = Menu(menu)
                self.addMenuItems(pullover, item[2])  # подменю:
                menu.add_cascade(label     = item[0], # создать подменю
                                 underline = item[1], # добавить каскад
                                 menu      = pullover)

    def makeToolBar(self):
        """
        создает панель с кнопками внизу, если необходимо
        expand=no, fill=x, чтобы ширина оставалась постоянной
        можно добавить поддержку изображений: смотрите главу 9,
        для чего придется создать минатюры в формате FIF или использовать
        расширение PIL
        """
        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, поэтому меню/панель инструментов
        всегда остаются вверху/внизу и обрезаются в последнюю очередь;
```

```
        переопределите этот метод,
        для pack: прикрепляйте середину к любому краю;
        для grid: компонуйте середину по сетке во фрейме, который
        прикрепляется методом pack
        """
        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):
        "переопределите в подклассе"
        showinfo('Help', 'Sorry, no help for ' + self.__class__.__name__)

    def start(self):
        "переопределите в подклассе: связать меню/панель инструментов с self"
        pass

################################################################################
# Специализированная версия для полосы меню главного окна Tk 8.0
################################################################################

GuiMakerFrameMenu = GuiMaker        # используется для меню встраиваемых
                                    # компонентов
class GuiMakerWindowMenu(GuiMaker): # используется для меню окна
    def makeMenuBar(self):          # верхнего уровня
        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) # В Linux требуется настоящее меню
                pulldown.add_command(label='About', command=self.help)
                menubar.add_cascade(label='Help', menu=pulldown)

################################################################################
# Реализация самотестирования, которая выполняется, если запустить модуль как
# самостоятельный сценарий: '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 использовать 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 принцип действия этого модуля, необходимо знакомство с основами создания меню, изложенными в главе 9. При соблюдении этого условия программный код будет прост и понятен: класс `GuiMaker` просто выполняет обход структур с описанием меню и панели инструментов и попутно создает соответствующие виджеты. В реализацию самотестирования этого модуля включен простой пример структур данных, использованных для компоновки меню и панели инструментов:

Шаблоны меню

Списки и вложенные подсписки кортежей (*метка*, *горячая_клавиша*, *обработчик*). Если *обработчик* является подсписком, а не функцией или методом, предполагается, что это каскадное подменю.

Шаблоны панелей инструментов

Список кортежей (*метка*, *обработчик*, *параметры_компоновки*). Параметры компоновки определяются в виде словаря параметров, передаваемых методу `pack` виджета, – словарь можно записать в виде литерала `{'k':v}` или использовать вызов функции `dict(k=v)` с именованными аргументами. Метод `pack` принимает словари, однако словари можно трансформировать в именованные аргументы, ис-

пользуя синтаксис вызова `func(**kargs)`. В данной реализации метки определяются как текст, но точно так же можно было бы реализовать поддержку изображений (смотрите раздел «BigGui: клиентская демонстрационная программа» ниже)

Для разнообразия предусмотрено изменение внешнего вида указателя мыши в зависимости от его местоположения: при наведении на панель инструментов указатель приобретает вид руки, в средней части – вид перекрестия, а при наведении на кнопку `Help` в меню, основанном на фрейме, – другой вид (можете настроить по вашему желанию).

Протоколы подклассов

Помимо структур меню и панелей инструментов клиенты этого класса могут вмешиваться и изменять реализованные в нем методы и протоколы компоновки:

Атрибуты шаблона

Предполагается, что клиенты этого класса установят атрибуты `menuBar` и `toolBar` в каком-то месте в цепочке наследования до момента завершения метода `start`.

Инициализация

Метод `start` может переопределяться для динамического создания шаблонов меню и панели инструментов, поскольку ему доступна ссылка `self`. Метод `start` служит также местом, где осуществляется общая инициализация – конструктор `__init__` класса `GuiMixin` должен вызываться, но не переопределяться.

Добавление виджетов

Метод `makeWidgets` может быть переопределен и создает виджеты в средней части окна – между полосой меню и панелью инструментов. По умолчанию `makeWidgets` помещает в середине метку с именем ближайшего класса, но по сути это абстрактный метод и предполагается его специализация в подклассах.

Протокол компоновки методом pack

В специализированном методе `makeWidgets` клиенты могут прикреплять виджеты средней части к любому краю `self` (`Frame`), так как полоса меню и панель инструментов уже захватили верх и низ контейнера к моменту выполнения `makeWidgets`. Если виджеты, составляющие среднюю часть, компонуются с помощью метода `pack`, она не обязательно должна быть вложенным фреймом. Полоса меню и панель инструментов автоматически компонуются первыми, чтобы при сжатии окна они обрезались в последнюю очередь.

Протокол компоновки методом grid

Размещение виджетов в средней части может осуществляться по сетке, если эта сетка помещена во вложенный фрейм, который до-

бавляется в родительский контейнер `self`. (Напомню, что на каждом уровне контейнеров можно применять любой из методов, `grid` или `pack`, но не оба вместе, а `self` является фреймом, в котором к моменту вызова `makeWidgets` меню и панель инструментов уже скомпонованы с применением метода `pack`.) Так как фрейм `GuiMaker` сам компонует себя в родительском контейнере с помощью метода `pack`, по аналогичным причинам его нельзя непосредственно встраивать в контейнер с элементами, располагаемыми по сетке, – для использования его в таком контексте добавьте промежуточный фрейм с сеткой.

Классы GuiMaker

В ответ на выполнение условий по протоколам и шаблонам `GuiMaker` клиентские подклассы получают фрейм, который умеет автоматически строить свои меню и панели инструментов по структурам данных шаблона. Если вы смотрели примеры создания меню в предыдущих главах, то сможете понять, что это большой шаг вперед, в смысле уменьшения объема программного кода. Класс `GuiMaker` также достаточно сообразителен и может экспортировать интерфейсы меню обоих стилей, с которыми мы встречались в главе 9:

`GuiMakerWindowMenu`

Реализует меню окон верхнего уровня в стиле Tk 8.0, которые удобно использовать в самостоятельных программах и всплывающих окнах.

`GuiMakerFrameMenu`

Реализует альтернативные меню, основанные на виджетах `Frame`/`Menubutton`, которые удобно использовать для создания меню объектов, встраиваемых в виде компонентов в более крупные графические интерфейсы.

Оба класса создают панели инструментов, экспортируют одни и те же протоколы и ожидают получить одни и те же структуры шаблонов – они отличаются только способом обработки шаблонов меню. В действительности один из них является подклассом другого, специализирующим метод создания меню – два стиля отличаются только обработкой меню верхнего уровня (`Menu` с каскадами `Menu`, вместо `Frame` с `Menubuttons`).

Программный код самотестирования GuiMaker

Как и в случае с классом `GuiMixin`, если запустить пример 10.3 как самостоятельный сценарий, будет выполнена логика самотестирования, находящаяся в конце файла. На рис. 10.2 изображены получаемые при этом окна. На экране создаются три окна, представляющие классы `TestApp`. Все три окна имеют меню и панель инструментов, параметры которых определены в структурах данных шаблонов, создаваемых программным кодом самотестирования: раскрывающиеся меню File и Edit,

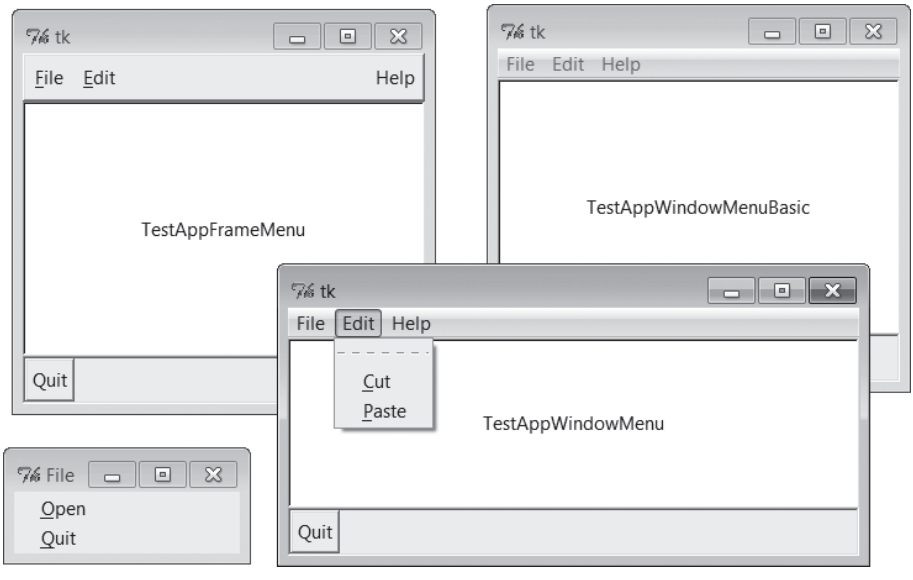


Рис. 10.2. *Программный код самотестирования GuiMaker в действии*

а также кнопка панели инструментов Quit и стандартная кнопка меню Help. На рисунке меню File одного из окон оторвано, а меню Edit другого окна раскрыто. Нижнее окно растянуто для наглядности.

Класс `GuiMaker` можно смешивать с другими суперклассами, но в первую очередь он предназначен служить тем же целям расширения и встраивания, что и класс `Frame` из библиотеки tkinter (особенно если учесть, что в действительности он является специализированным классом `Frame`, реализующим дополнительные протоколы конструирования). Фактически программный код самотестирования объединяет фрейм `GuiMaker` с инструментами из класса `GuiMixin`, представленного в предыдущем разделе.

Связи между суперклассами устанавливаются в программном коде так, что два из трех окон получают обработчик `help` из класса `GuiMixin`, а `TestAppWindowMenuBasic` получает его из класса `GuiMaker`. Обратите внимание, что порядок, в котором смешиваются эти два класса, имеет большое значение: 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 обеспечить преимущество методов класса `GuiMixin`, его следует указывать перед суперклассом, производным от виджетов.

Более практическое применение класс `GuiMaker` найдет в таких примерах, как PyEdit в главе 11. В следующем разделе демонстрируется

другой способ использования шаблонов класса GuiMaker для построения усложненного интерфейса, который служит еще одной проверкой его функциональных возможностей.

BigGui: клиентская демонстрационная программа

Рассмотрим программу, представляющую лучшее применение тех двух классов автоматизации, которые мы написали. Класс Hello, реализованный в примере 10.4, является наследником обоих классов, GuiMixin и GuiMaker. Класс GuiMaker обеспечивает связь с виджетом Frame и логику создания меню/панели инструментов. Класс GuiMixin обеспечивает дополнительные методы стандартного поведения. В действительности класс Hello служит образцом еще одного способа расширения виджета Frame, поскольку является производным от класса GuiMaker. Чтобы бесплатно получить меню и панель инструментов, он просто следует протоколам, определенным в классе GuiMaker, – устанавливает атрибуты menuBar и toolBar в методе start и переопределяет метод makeWidgets, помещая в середину нестандартную метку.

Пример 10.4. PP4E\Gui\Tools\big_gui.py

```
"""
реализация графического интерфейса - объединяет GuiMaker, GuiMixin и данный
класс
"""

import sys, os
from tkinter import *                    # классы виджетов
from PP4E.Gui.Tools.guimixin import *    # подмешиваемые методы: quit, spawn...
from PP4E.Gui.Tools.guimaker import *    # фрейм плюс построение меню/панели
                                         # инструментов
class Hello(GuiMixin, GuiMakerWindowMenu):      # или GuiMakerFrameMenu
    def start(self):
        self.hellos = 0
        self.master.title("GuiMaker Demo")
        self.master.iconname("GuiMaker")
        def spawnme(): self.spawn('big_gui.py') # отложен. вызов вместо lambda

        self.menuBar = [                        # дерево: 3 раскр. меню
            ('File', 0,                         # (раскр. меню)
                [('New...', 0, spawnme),
                 ('Open...', 0, self.fileOpen), # [список элементов меню]
                 ('Quit', 0, self.quit)]        # метка, клавиша, обработчик
            ),

            ('Edit', 0,
                [('Cut', -1, self.notdone),     # без клавиши| обработчика
                 ('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,
                 [('Toplevels', 0,
                    lambda: self.spawn(r'..\Tour\toplevel2.py')),
                  ('Frames', 0,
                    lambda: self.spawn(r'..\Tour\demoAll-frm-ridge.py')),
                  ('Images', 0,
                    lambda: self.spawn(r'..\Tour\buttonpics.py')),
                  ('Alarm', 0,
                    lambda: self.spawn(r'..\Tour\alarm.py', wait=False)),
                  ('Other...', -1, self.pickDemo)]
            )]
        )]

    self.toolBar = [                           # добавить 3 кнопки
        ('Quit', self.quit, dict(side=RIGHT)), # или {'side': RIGHT}
        ('Hello', self.greeting, dict(side=LEFT)),
        ('Popup', self.dialog, dict(side=LEFT, expand=YES)) ]

def makeWidgets(self):                                 # переопределить метод
    middle = Label(self, text='Hello maker world!', # создания виджетов
                   width=40, height=10,             # в середине окна
                   relief=SUNKEN, cursor='pencil', bg='white')
    middle.pack(expand=YES, fill=BOTH)

def greeting(self):
    self.hellos += 1
    if self.hellos % 3:
        print("hi")
    else:
        self.infobox("Three", 'HELLO!')     # каждый третий щелчок

def dialog(self):
    button = self.question('OOPS!',
                           'You typed "rm*" ... continue?', # старый стиль
                           'questhead', ('yes', 'no'))      # аргументы
    [lambda: None, self.quit][button]()                     # игнорируются

def fileOpen(self):
    pick = self.selectOpenFile(file='big_gui.py')
```

```
            if pick:
                self.browser(pick)     # просмотр файла модуля или другого файла

        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='..')
            if pick:
                self.spawn(pick)       # запустить любую программу Python

    if __name__ == '__main__': Hello().mainloop()   # создать, запустить
```

Этот сценарий создает довольно объемное меню и панель инструментов, а также добавляет собственные методы обратного вызова, которые выводят сообщения в поток `stdout`, отображают средства просмотра текстовых файлов и новые окна и запускают другие программы. Однако многие из этих методов не делают ничего, кроме запуска метода `notDone`, унаследованного от класса `GuiMixin`. Данный пример предназначен в основном для демонстрации возможностей классов `GuiMaker` и `GuiMixin`.

Если запустить `big_gui` как самостоятельный сценарий, он создаст окно с четырьмя раскрывающимися меню вверху и панелью инструментов с тремя кнопками внизу, как показано на рис. 10.3, где также изображены некоторые всплывающие окна, созданные обработчиками. В меню имеются разделители, неактивные элементы и каскадные подменю в полном соответствии с шаблоном `menuBar`, который передается классу `GuiMaker`, и кнопка Quit, унаследованная от класса `GuiMixin`, щелчок на которой вызывает появление диалога с просьбой подтвердить завершение работы. И это лишь часть инструментов, которые мы бесплатно получаем в свое распоряжение.

На рис. 10.4 снова изображено окно этого сценария после того как через раскрывающееся меню Play были запущены два демонстрационных сценария, которые мы написали в главах 8 и 9, выполняющиеся независимо. Эти демонстрационные сценарии были запущены с помощью переносимых инструментов, которые мы написали в главе 5 и приобрели из класса `GuiMixin`. Если у вас появится желание запустить какую-либо другую демонстрационную программу, выберите пункт Other в меню Play, который откроет стандартный диалог открытия файла, и выберите файл требуемой программы. Примечание: изображение ярлыка, используемое демонстрационным сценарием, запуск которого производится из меню Play, я скопировал в каталог с этим сценарием – позднее мы напишем инструменты, которые будут пытаться отыскивать его автоматически.

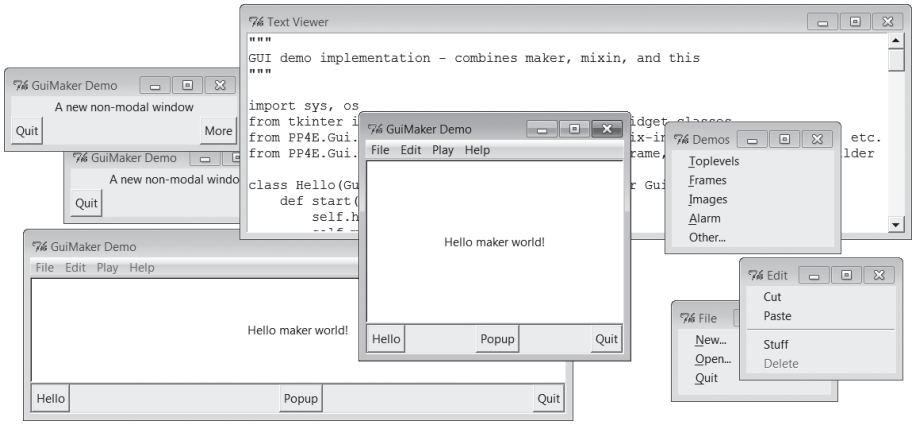


Рис. 10.3. *Сценарий big_gui с несколькими всплывающими окнами*

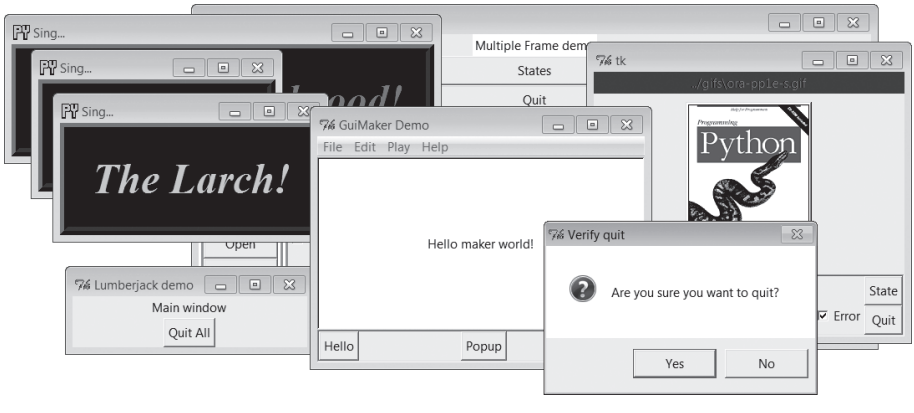


Рис. 10.4. *Сценарий big_gui и несколько запущенных им демонстрационных программ*

Наконец, хочу заметить, что класс `GuiMaker` можно перепроектировать так, чтобы в нем использовались деревья вложенных экземпляров классов, умеющих применять себя к строящемуся дереву виджетов tkinter вместо ветвления по типам элементов в структурах данных шаблона. Однако ввиду недостатка места в данном издании мы отнесем это расширение к числу самостоятельных упражнений.

Коль скоро речь зашла о расширениях, – в главе 9 я демонстрировал, как вместо текстовых меток помещать на кнопки в панели инструментов изображения. Создание подкласса, наследующего класс `GuiMaker`, реализующего такую возмож-

ность за счет переопределения метода создания панели инструментов, стало бы не только отличным упражнением, но и позволило бы получить полезный инструмент. Однако если я буду добавлять реализацию каждой такой особенности, эта книга может разрастись настолько, что превратится в неподъемный груз...

ShellGui: графические интерфейсы к инструментам командной строки

Демонстрационные программы – это здорово, но, чтобы явственнее продемонстрировать практическую пользу таких инструментов, как класс `GuiMixin`, требуется найти ему более реальное применение. Вот одно из них: допустим, что мы написали комплект сценариев командной строки для решения административных задач, например, таких, как мы написали во второй части книги. Там, если помните, сценарии запускались из командной строки, но требовали от нас запоминать все параметры, которые можно передавать им при запуске, – для меня (и, вероятно, для вас) это означает, что при обращении к сценариям после долгого перерыва потребуется детально изучить их программный код.

Вместо того чтобы требовать от пользователей таких инструментов вводить загадочные команды в оболочке, почему бы для запуска этих программ не создать простой в использовании графический интерфейс на базе tkinter? Такой интерфейс мог бы запрашивать параметры командной строки, не требуя, чтобы пользователь помнил их. И если мы ставим такую задачу, почему бы вообще не обобщить идею запуска инструментов командной строки из графического интерфейса для обеспечения поддержки инструментов, которые появятся в будущем?

Обобщенный графический интерфейс инструментов оболочки

Примеры с 10.5 по 10.11 – два сценария командной строки, один вспомогательный модуль для реализации графического интерфейса, два диалога, главный графический интерфейс и модуль формы ввода параметров – представляют 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 углубления в иерархию программного обеспечения. А на самом верху все должно быть максимально универсальным, как показано в примере 10.5.

Пример 10.5. PP4E\Gui\ShellGui\shellgui.py

```
#!/usr/local/bin/python
"""
###############################################################################
инструмент запуска; использует шаблоны GuiMaker, стандартный диалог завершения
GuiMixin; это просто библиотека классов: чтобы вывести графический интерфейс,
запустите сценарий mytools;
###############################################################################
"""

from tkinter import *                           # импортировать виджеты
from PP4E.Gui.Tools.guimixin import GuiMixin    # импортировать quit, а не done
from PP4E.Gui.Tools.guimaker import *           # конструктор меню/панели
                                                # инструментов
class ShellGui(GuiMixin, GuiMakerWindowMenu):  # фрейм + конструктор +
    def start(self):                            # подмешиваемые методы
        self.setMenuBar()                       # для компонентов использовать
        self.setToolBar()                       # GuiMaker
        self.master.title("Shell Tools Listbox")
        self.master.iconname("Shell Tools")

    def handleList(self, event):                # двойной щелчок на списке
        label = self.listbox.get(ACTIVE)        # получить выбранный текст
        self.runCommand(label)                  # и выполнить операцию

    def makeWidgets(self):                      # добавить список в середину
        sbar = Scrollbar(self)                  # связать sbar со списком
        list = Listbox(self, bg='white')        # или использ. Tour.ScrolledList
        sbar.config(command=list.yview)
        list.config(yscrollcommand=sbar.set)
        sbar.pack(side=RIGHT, fill=Y)           # первым добавлен = посл. обрезан
        list.pack(side=LEFT, expand=YES, fill=BOTH)  # список обрез-ся первым
        for (label, action) in self.fetchCommands(): # добавляется в список,
            list.insert(END, label)                  # в меню и на панель инстр.
        list.bind('<Double-1>', self.handleList)     # установить обработчик
        self.listbox = list

    def forToolBar(self, label):                # поместить на панель инстр.?
        return True                             # по умолчанию = все

    def setToolBar(self):
        self.toolBar = []
        for (label, action) in self.fetchCommands():
            if self.forToolBar(label):
                self.toolBar.append((label, action, dict(side=LEFT)))
        self.toolBar.append(('Quit', self.quit, dict(side=RIGHT)))

    def setMenuBar(self):
```

```
        toolEntries = []
        self.menuBar = [
            ('File', 0, [('Quit', -1, self.quit)]), # имя раскрывающегося меню
            ('Tools', 0, toolEntries)               # список элементов меню
        ]                                           # метка, клавиша, обработчик
        for (label, action) in self.fetchCommands():
            toolEntries.append((label, -1, action)) # добавить приложения
                                                    # в меню
################################################################################
# делегирование операций шаблонным подклассам с разным способом хранения
# перечня утилит, которые в свою очередь делегируют операции
# подклассам, реализующим запуск утилит
################################################################################

class ListMenuGui(ShellGui):
    def fetchCommands(self):           # myMenu устанавливается в подклассе
        return self.myMenu             # список кортежей (метка, обработчик)
    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, знает, как с помощью интерфейсов `GuiMaker` и `GuiMixin` создать окно для выбора, которое выводит имена утилит в меню, в списке с прокруткой и на панели инструментов. Он также предоставляет переопределяемый метод `forToolBar`, позволяющий подклассам указывать, какие утилиты должны добавляться на панель инструментов, а какие – нет (на панели инструментов может быстро закончиться свободное место). Однако он умышленно оставлен в неведении относительно имен утилит, которые должны быть выведены в указанных местах, и операций, которые должны быть выполнены при выборе имен утилит.

Вместо этого класс `ShellGui` использует подклассы `ListMenuGui` и `DictMenuGui`, находящиеся в этом же файле, чтобы получить список имен утилит через их методы `fetchCommands` и управлять операциями по именам с помощью их методов `runCommand`. Эти два подкласса в действительности лишь предоставляют интерфейс к наборам утилит, представленным в виде списков и словарей, – они по-прежнему не знают, какие имена утилит будут реально отображены в графическом интерфейсе. Это сделано умышленно: так как отображаемые наборы утилит определяются подклассами более низкого уровня, мы получаем возможность использовать класс `ShellGui` для отображения различных наборов утилит.

Классы наборов утилит

Чтобы получить фактические наборы утилит, нужно спуститься на один уровень ниже. Модуль в примере 10.6 определяет подклассы двух специфических по типу классов, наследующих класс `ShellGui`, чтобы предоставить наборы доступных инструментов в виде списка и словаря (обычно достаточно одного из них, но модуль иллюстрирует применение обоих). Кроме того, именно этот модуль *запускает* графический интерфейс – модуль `shellgui` является всего лишь библиотекой классов.

Пример 10.6. PP4E\Gui\ShellGui\mytools.py

```
#!/usr/local/bin/python
"""
###############################################################################
реализует два набора инструментов, специфичных для типов
###############################################################################
"""

from shellgui import *              # интерфейсы, специфичные для типов
from packdlg import runPackDialog   # диалоги для ввода данных
from unpkdlg import runUnpackDialog # оба используют классы приложений

class TextPak1(ListMenuGui):
    def __init__(self):
        self.myMenu = [('Pack  ', runPackDialog),   # простые функции
                       ('Unpack', runUnpackDialog), # длина меток одинаковая
                       ('Mtool ', self.notdone)]    # метод из GuiMixin
        ListMenuGui.__init__(self)
    def forToolBar(self, label):
        return label in {'Pack  ', 'Unpack'}        # синтаксис множеств в 3.x

class TextPak2(DictMenuGui):
    def __init__(self):
        self.myMenu = {'Pack  ': runPackDialog,   # или использовать input...
                       'Unpack': runUnpackDialog, # вместо диалогов ввода
                       'Mtool ': self.notdone}
        DictMenuGui.__init__(self)

if __name__ == '__main__':                        # реализация 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 являются конкретными наборами утилит. Чтобы вывести другой набор имен утилит, нужно просто написать и использовать новый подкласс. Разделение логики приложения на такие отдель-

ные подклассы и модули повышает возможность повторного использования программного обеспечения.

На рис. 10.5 изображено главное окно `ShellGui`, создаваемое при запуске сценария `mytools` с классом структуры меню на основе списка в Windows 7, а также оторванные меню, демонстрирующие свое содержание. Меню и панель инструментов этого окна построены с помощью класса `GuiMaker`, а кнопки `Quit` и `Help` и пункты меню, вызывающие методы `quit` и `help`, унаследованы из класса `GuiMixin` через суперклассы `ShellGui` модуля. Надеюсь, вы начинаете понимать, почему в этой книге столь часто проповедуется повторное использование программного кода?

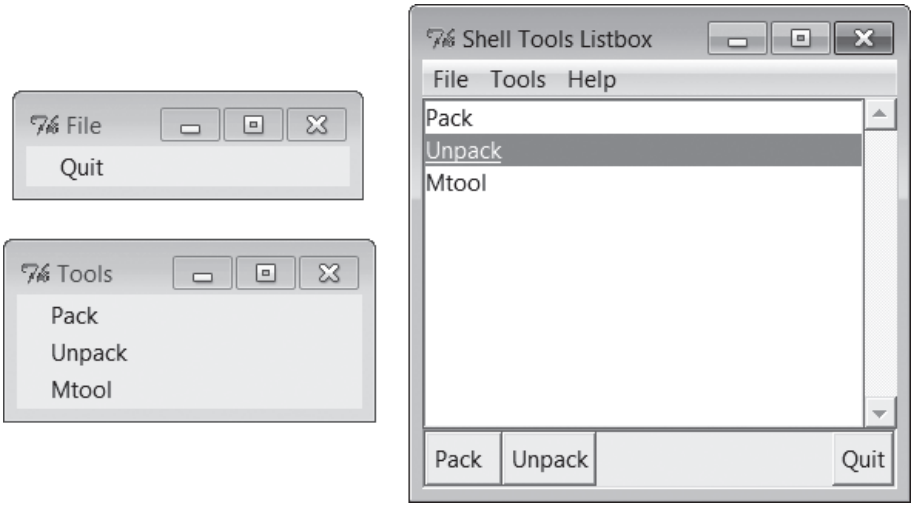


***Рис. 10.5.** Элементы mytools в окне ShellGui*

Добавление графических интерфейсов к инструментам командной строки

К настоящему моменту мы создали библиотеку универсальных инструментальных классов, а также модуль с комплектом инструментов для запуска конкретных приложений, который определяет имена обработчиков в пунктах меню. Чтобы закончить картину, нам необходимо теперь реализовать эти обработчики, а также сами сценарии, которые они будут запускать.

Сценарии командной строки

Для проверки способности графического интерфейса запускать сценарии командной строки нам, конечно же, необходимо создать несколько таких сценариев. Следующие два сценария, находящиеся в самом низу иерархии, реализуют архивацию текстовых файлов, используя

системные инструменты и приемы, описанные во второй части книги. Первый сценарий, представленный в примере 10.7, просто объединяет содержимое нескольких текстовых файлов в один файл, добавляя предопределенные строки-разделители между ними.

Пример 10.7. PP4E\Gui\ShellGui\packer.py

```
# упаковывает текстовые файлы в единый файл, добавляя строки-разделители
# (простейшая архивация)

import sys, glob
marker = ':' * 20 + 'textpak=>'    # надеемся, что это уникальная строка

def pack(ofile, ifiles):
    output = open(ofile, 'w')
    for name in ifiles:
        print('packing:', name)
        input = open(name, 'r').read()      # открыть следующий входной файл
        if input[-1] != '\n': input += '\n' # гарантировать наличие \n в конце
        output.write(marker + name + '\n')  # записать строку-разделитель
        output.write(input)                 # и содержимое входного файла

if __name__ == '__main__':
    ifiles = []
    for patt in sys.argv[2:]:     # в Windows не выполняется автоматическая
        ifiles += glob.glob(patt) # подстановка по шаблону
    pack(sys.argv[1], ifiles)     # упаковать файлы, перечисленные
                                  # в командной строке
```

Второй сценарий, который приводится в примере 10.8, сканирует файлы архивов, созданные первым сценарием, и восстанавливает оригинальные файлы.

Пример 10.8. PP4E\Gui\ShellGui\unpacker.py

```
# распаковывает архивы, созданные сценарием packer.py
# (простейшие архивы текстовых файлов)

import sys
from packer import marker # использовать общую строку-разделитель
mlen = len(marker)        # имена файлов следуют за строкой-разделителем

def unpack(ifile, prefix='new-'):
    for line in open(ifile):              # по всем строкам входного файла
        if line[:mlen] != marker:
            output.write(line)            # действительные строки записать
        else:
            name = prefix + line[mlen:-1] # или создать новый выходной файл
            print('creating:', name)
            output = open(name, 'w')

if __name__ == '__main__': unpack(sys.argv[1])
```

Это чрезвычайно простые сценарии, и в этой части книги предполагается, что вы уже прочитали главы, посвященные системным инструментам, поэтому мы не будем вдаваться в подробности их реализации. Варианты этих сценариев появились еще в первом издании этой книги, в 1996 году. Я пользовался ими на заре моей карьеры программиста на языке Python для архивации файлов, еще до того, как на всех моих компьютерах появились такие инструменты, как tar и zip (и еще до того, как в стандартной библиотеке Python появились модули поддержки tar и zip). Принцип действия этих сценариев чрезвычайно прост. Возьмем следующие три текстовых файла:

```
C:\...\PP4E\Gui\ShellGui> type spam.txt
spam
Spam
SPAM
C:\...\PP4E\Gui\ShellGui> type eggs.txt
eggs

C:\...\PP4E\Gui\ShellGui> type ham.txt
h
  a
    m
```

Если запустить сценарий `packer` из командной строки, он объединит эти файлы в один общий файл, а сценарий `unpacker` извлечет их оттуда. Сценарий `packer` должен предусматривать обработку шаблонов имен файлов, потому что командная оболочка в Windows не выполняет автоматическое расширение шаблонов:

```
C:\...\PP4E\Gui\ShellGui> packer.py packed.txt *.txt
packing: eggs.txt
packing: ham.txt
packing: spam.txt

C:\...\PP4E\Gui\ShellGui> unpacker.py packed.txt
creating: new-eggs.txt
creating: new-ham.txt
creating: new-spam.txt
```

Файлы, извлекаемые из архива, по умолчанию получают уникальные имена (с дополнительным префиксом, чтобы избежать случайного затирания оригинальных файлов, что особенно важно на этапе тестирования), и вы получаете то, что было упаковано в архив:

```
C:\...\PP4E\Gui\ShellGui> type new-spam.txt
spam
Spam
SPAM

C:\...\PP4E\Gui\ShellGui> type packed.txt
::::::::::::::::::::textpak=>eggs.txt
```

```
eggs
::::::::::::::::::::textpak=>ham.txt
h
  a
    m
::::::::::::::::::::textpak=>spam.txt
spam
Spam
SPAM
```

Эти сценарии не предназначены для архивации двоичных файлов, не выполняют сжатие или что-то еще, а служат лишь иллюстрацией сценариев командной строки с обязательными аргументами командной строки. Они могут использоваться самостоятельно, как было показано выше (и запускаться с помощью таких инструментов в языке Python, как `os.popen` и `subprocess`), или как модули, которые могут импортироваться и вызываться другими программами. В нашем графическом интерфейсе мы будем использовать второй, более прямой интерфейс вызовов.

Диалоги ввода

Нам осталось реализовать заключительную часть. Сценарии упаковывания и распаковывания прекрасно справляются со своей работой как инструменты командной строки. Однако обработчики, имена которых указаны в сценарии *mytools.py* из примера 10.6, должны делать нечто, ориентированное на использование графического интерфейса. Поскольку оригинальные сценарии `packer` и `unpacker` живут в мире текстовых потоков ввода-вывода и командных оболочек, нам необходимо обернуть их программным кодом, который будет принимать входные параметры из графического интерфейса. В частности, нам необходимы диалоги, запрашивающие обязательные аргументы командной строки.

В первую очередь рассмотрим модуль, представленный в примере 10.9, и клиентский сценарий в примере 10.10, который использует приемы создания модального диалога, рассматривавшиеся в главе 8, чтобы отобразить форму ввода параметров для сценария `packer`. Программный код в примере 10.9 был выделен в отдельный модуль, потому что он может найти более широкое применение. Фактически мы будем повторно использовать его в реализации диалога для сценария `unpacker` и еще раз – в приложении PyEdit, в главе 11.

Этот модуль демонстрирует еще один способ автоматизации конструирования графических интерфейсов – его использование для создания рядов формы ввода позволяет заменить 7 или более строк программного кода для каждого ряда (6 – если не использовать связанную переменную или кнопку вызова диалога выбора файла) ровно на 1 строку. В модуле *form.py*, в главе 12, мы увидим другой, еще более автоматизированный способ конструирования форм. Однако уже такой автомати-

зации вполне достаточно, чтобы сэкономить десятки строк программного кода при создании нетривиальных форм.

Пример 10.9. PP4E\Gui\ShellGui\formrows.py

```
"""
создает фрейм-ряд с меткой и полем ввода и дополнительной кнопкой, вызывающей
диалог выбора файла; эта реализация была выделена в отдельный модуль, потому что
она может с успехом использоваться и в других программах; вызывающая программа
(или обработчики событий, как в данном случае) должна сохранять ссылку на
связанную переменную на все время использования ряда;
"""

from tkinter import *                           # виджеты и константы
from tkinter.filedialog import askopenfilename # диалог выбора файла

def makeFormRow(parent, label, width=15, browse=True, extend=False):
    var = StringVar()
    row = Frame(parent)
    lab = Label(row, text=label + '?', relief=RIDGE, width=width)
    ent = Entry(row, relief=SUNKEN, textvariable=var)
    row.pack(fill=X)                            # используются фреймы-ряды
    lab.pack(side=LEFT)                         # с метками фиксированной длины
    ent.pack(side=LEFT, expand=YES, fill=X)     # можно использовать
    if browse:                                  # grid(row, col)
        btn = Button(row, text='browse...')
        btn.pack(side=RIGHT)
        if not extend:
            btn.config(command=
                lambda: var.set(askopenfilename() or var.get()) )
        else:
            btn.config(command=
            lambda: var.set(var.get() + ' ' + askopenfilename()) )
    return var
```

Далее, функция `runPackDialog` в примере 10.10 является фактическим обработчиком, который вызывается при выборе имени инструмента в главном окне `ShellGui`. Она использует модуль конструирования рядов формы из примера 10.9 и применяет приемы создания модальных диалогов, которые мы изучали ранее.

Пример 10.10. PP4E\Gui\ShellGui\packdlg.py

```
# выводит диалог ввода параметров для сценария packer и запускает его

from glob import glob               # расширение шаблонов имен файлов
from tkinter import *               # виджеты графического интерфейса
from packer import pack             # использовать сценарий/модуль packer
from formrows import makeFormRow    # использовать инструмент создания форм

def packDialog():                   # новое окно верхнего уровня
    win = Toplevel()                # с 2 фреймами-рядами + кнопка ok
```

```
    win.title('Enter Pack Parameters')
    var1 = makeFormRow(win, label='Output file')
    var2 = makeFormRow(win, label='Files to pack', extend=True)
    Button(win, text='OK', command=win.destroy).pack()
    win.grab_set()
    win.focus_set()               # модальный: захватить мышь, фокус ввода,
    win.wait_window()             # ждать закрытия окна диалога;
                                  # иначе возврат произойдет немедленно
    return var1.get(), var2.get()  # извлечь значения связанных переменных

def runPackDialog():
    output, patterns = packDialog()      # вывести диалог и ждать щелчка на
    if output != "" and patterns != "":  # кнопке ok или закрытия окна
        patterns = patterns.split()      # выполнить действия не связанные с
        filenames = []                   # графическим интерфейсом
        for sublist in map(glob, patterns):  # вып. расширение шаблона вручную
            filenames += sublist             # командные оболочки Unix
        print('Packer:', output, filenames)  # делают это автоматически
        pack(ofile=output, ifiles=filenames) # вывод также можно показать в
                                             # графическом интерфейсе
if __name__ == '__main__':
    root = Tk()
    Button(root, text='popup', command=runPackDialog).pack(fill=X)
    Button(root, text='bye', command=root.quit).pack(fill=X)
    root.mainloop()
```

Если запустить сценарий из примера 10.10 и щелкнуть на кнопке popup, он создаст форму ввода, как показано на рис. 10.6, – это тот же диалог, который будет показан в ответ на выбор инструмента в главном окне сценария *mytools.py*. Пользователь может ввести имена входных и выходных файлов с клавиатуры или щелкнуть на кнопке browse... чтобы открыть стандартный диалог выбора файла. Допускается вводить шаблоны имен файлов – вызов функции glob в этом сценарии выполнит подстановку шаблона и отфильтрует имена несуществующих файлов. Командные оболочки в Unix осуществляют такую подстановку шаблонов автоматически, если запускать сценарий *packer.py* из командной строки, в отличие от Windows.

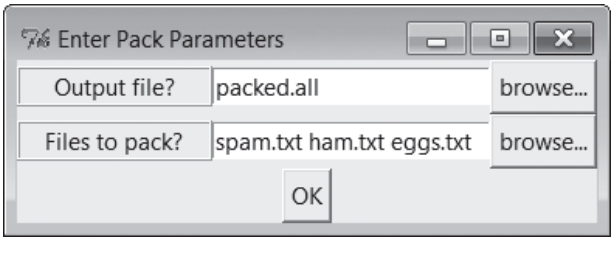


***Рис. 10.6.** Форма ввода packdlg*

После того как пользователь заполнит форму и щелкнет на кнопке OK, параметры будут переданы главной функции сценария packer, представленного выше, для выполнения операции слияния файлов.

Графический интерфейс диалога ввода параметров для сценария unpacking выглядит проще, потому что в нем присутствует только одно поле ввода – имя файла архива. Здесь мы снова используем модуль конструирования рядов формы ввода, разработанного для диалога к сценарию packer, потому что эти две задачи очень похожи. Сценарий в примере 10.11 (и его главная функция, вызываемая графическим интерфейсом выбора инструмента в сценарии *mytools.py*) создает форму ввода, изображенную на рис. 10.7.

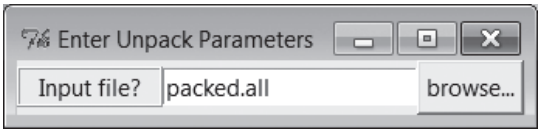


***Рис. 10.7.** Форма ввода unpkdlg*

Пример 10.11. PP4E\Gui\ShellGui\unpkdlg.py

```
# выводит диалог ввода параметров для сценария unpacker и запускает его

from tkinter import *              # классы виджетов
from unpacker import unpack        # использовать сценарий/модуль unpacker
from formrows import makeFormRow   # инструмент создания полей формы

def unpackDialog():
    win = Toplevel()
    win.title('Enter Unpack Parameters')
    var = makeFormRow(win, label='Input file', width=11)
    win.bind('<Key-Return>', lambda event: win.destroy())
    win.grab_set()
    win.focus_set()                # сделать себя модальным
    win.wait_window()              # ждать возврата из диалога
    return var.get()               # или закрытия его окна

def runUnpackDialog():
    input = unpackDialog()             # получить входные параметры из диалога
    if input != '':                    # выполнить действия, не связанные с
        print('Unpacker:', input)      # графическим интерфейсом, передав имя
        unpack(ifile=input, prefix='') # файла из диалога

if __name__ == "__main__":
    Button(None, text='popup', command=runUnpackDialog).pack()
    mainloop()
```

Кнопка browse... на рис. 10.7 выводит диалог выбора файла так же, как форма packdlg. Вместо кнопки OK этот диалог связывает событие нажа-

тия клавиши Enter с операцией закрытия окна и с завершением ожидания закрытия модального диалога; при этом имя файла архива передается экземпляру класса главной функции сценария unpacker, представленного выше, для выполнения фактической процедуры сканирования файла.

Возможные улучшения

Весь комплекс действует, как и было обещано, – благодаря такой доступности утилит командной строки в графическом виде они становятся значительно более привлекательными для пользователей, привычной средой обитания которых является графический интерфейс. Мы фактически добавили простой графический интерфейс к инструментам командной строки. И все же в данной конструкции есть два аспекта, которые можно было бы усовершенствовать.

Во-первых, оба диалога ввода используют общий программный код конструирования рядов в их формах ввода, который был реализован для данного конкретного случая. Мы могли бы существенно упростить создание диалогов, импортировав более обобщенный модуль создания форм. Мы встречались с обобщенной реализацией конструктора форм в главах 8 и 9 и будем встречаться с ней далее – смотрите также указания по обобщению создания форм в модуле form.py, в главе 12.

Во-вторых, в тот момент, когда пользователь передает данные, введенные в той или иной диалоговой форме, след графического интерфейса теряется – главное окно блокируется, а сообщения поступают в окно консоли. Графический интерфейс блокируется по техническим причинам и не может обновлять себя, пока работают утилиты packer и unpacker. Хотя эти операции и выполняются достаточно быстро с моими файлами, но при обработке очень больших файлов можно было бы запускать их в отдельных потоках выполнения, чтобы сохранить графический интерфейс активным (подробнее о потоках выполнения рассказывается далее в этой главе).

Проблема с консолью является более насущной: сообщения от утилит packer и unpacker все так же выводятся в поток stdout, а не в графический интерфейс (все имена файлов здесь включают полные пути к ним, если выбирать их с помощью стандартного диалога выбора файлов, открываемого щелчком на кнопке browse...):

```
C:\...\PP4E\Gui\ShellGui\temp> python ..\mytools.py list
PP4E scrolledtext
list test
Packer: packed.all ['spam.txt', 'ham.txt', 'eggs.txt']
packing: spam.txt
packing: ham.txt
packing: eggs.txt
Unpacker: packed.all
creating: spam.txt
```

```
creating: ham.txt
creating: eggs.txt
```

Это далеко не идеальное решение для пользователей, привыкших работать с графическим интерфейсом, – они могут не ожидать (или даже не в состоянии отыскать) появления полезной информации в окне консоли. Лучшим решением было бы *перенаправить* поток вывода `stdout` в объект, который выводит полученный текст в окно графического интерфейса. О том, как это сделать, можно прочесть в следующем разделе.

GuiStreams: перенаправление потоков данных в виджеты

Следующий прием программирования графических интерфейсов. В ответ на проблему, поставленную в конце предыдущего раздела, сценарий в примере 10.12 организует отображение входных и выходных потоков во всплывающие окна приложения, применяя практически тот же способ, который мы использовали со строками в темах, связанных с перенаправлением потоков ввода-вывода в главе 3. Хотя этот модуль служит лишь базовым прототипом и сам нуждается в усовершенствовании (например, для ввода каждой входной строки отображается новый диалог – не самое эргономичное решение), тем не менее он демонстрирует идею в целом.

Объекты этого модуля, `GuiOutput` и `GuiInput`, определяют методы, позволяющие им маскироваться под файлы везде, где ожидаются настоящие файлы. Как мы узнали в главе 3, это осуществляется с помощью средств доступа к стандартным потокам ввода-вывода, таких как встроенные функции `print` и `input`, и явных вызовов `read` и `write`. Типичные случаи использования обслуживаются в этом модуле двумя высокоуровневыми интерфейсами:

- Функция `redirectedGuiFunc` благодаря такой совместимости с файлами позволяет выполнять любые функции так, что их стандартные потоки ввода и вывода целиком отображаются в окна графического интерфейса, а не в окно консоли (или туда, куда эти потоки отображались бы в обычном случае).
- Функция `redirectedGuiShellCmd` аналогичным образом направляет в окно графического интерфейса вывод программы, запускаемой из командной строки. Она может использоваться для отображения в графическом интерфейсе вывода любых программ, включая программы на языке Python.

Классы `GuiInput` и `GuiOutput`, реализованные в модуле, можно использовать или специализировать непосредственно в клиентах, где требуется обеспечить более непосредственный интерфейс методов файлов или более полное управление процессом.

Пример 10.12. PP4E\Gui\Tools\guiStreams.py

```
"""
################################################################################
начальная реализация классов, похожих на файлы, которые можно использовать для
перенаправления потоков ввода и вывода в графические интерфейсы; входные данные
поступают из стандартного диалога (единый интерфейс вывод+ввод или постоянное
поле Entry для ввода были бы удобнее); кроме того, некорректно берутся строки
в запросах входных данных, когда количество байтов > len(строки); в GuiInput
можно было бы добавить методы __iter__/__next__, для поддержки итераций по
строкам, как в файлах, но это способствовало бы порождению большого количества
всплывающих окон;
################################################################################
"""

from tkinter import *
from tkinter.simpledialog import askstring
from tkinter.scrolledtext import ScrolledText # или PP4E.Gui.Tour.scrolledtext

class GuiOutput:
    font = ('courier', 9, 'normal')   # в классе - для всех, self - для одного
    def __init__(self, parent=None):
        self.text = None
        if parent: self.popupnow(parent) # сейчас или при первой записи

    def popupnow(self, parent=None):     # сейчас в родителе, Toplevel потом
        if self.text: return
        self.text = ScrolledText(parent or Toplevel())
        self.text.config(font=self.font)
        self.text.pack()

    def write(self, text):
        self.popupnow()
        self.text.insert(END, str(text))
        self.text.see(END)
        self.text.update()                  # обновлять после каждой строки

    def writelines(self, lines):            # строки уже включают '\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 ''                       # диалог для ввода каждой строки
        else:                               # кнопка cancel означает eof
            return line + '\n'              # иначе добавить символ '\n'
```

```
    def read(self, bytes=None):
        if not self.buff:
            self.buff = self.inputLine()
        if bytes:                           # читать по счетчику байтов,
            text = self.buff[:bytes]        # чтобы не захватить лишние строки
            self.buff = self.buff[bytes:]
        else:
            text = ''                       # читать 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() # имитировать методы чтения файла
        self.buff = ''
        return text

    def readlines(self):
        lines = []                          # читать все строки
        while True:
            next = self.readline()
            if not next: break
            lines.append(next)
        return lines

def redirectedGuiFunc(func, *pargs, **kargs):
    import sys                              # отображает потоки функции
    saveStreams = sys.stdin, sys.stdout     # во всплывающие окна
    sys.stdin   = GuiInput()                # выводит диалог при необходимости
    sys.stdout  = GuiOutput()               # новое окно для каждого вызова
    sys.stderr  = sys.stdout
    result = func(*pargs, **kargs)          # это блокирующий вызов
    sys.stdin, sys.stdout = saveStreams
    return result

def redirectedGuiShellCmd(command):
import os
    input = os.popen(command, 'r')
    output = GuiOutput()
    def reader(input, output):              # показать стандартный вывод
        while True:                         # команды оболочки в новом
            line = input.readline()         # окне с виджетом Text;
            if not line: break              # вызов readline может
            output.write(line)              # блокироваться
    reader(input, output)

if __name__ == '__main__':                  # код самотестирования
    def makeUpper():                        # использовать стандартные потоки
        while True:                         # ввода-вывода
```

```
            try:
                line = input('Line? ')
            except:
                break
            print(line.upper())
        print('end of file')

    def makeLower(input, output):           # использовать файлы
        while True:
            line = input.readline()
            if not line: break
            output.write(line.lower())
        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: redirectedGuiShellCmd('dir *')).pack(fill=X)
    root.mainloop()
```

Класс `GuiOutput` прикрепляет объект `ScrolledText` (из стандартной библиотеки Python) либо к указанному родительскому контейнеру, либо выводит новое окно верхнего уровня, которое должно служить контейнером, при первом обращении к методу записи. Класс `GuiInput` выводит новый стандартный диалог ввода каждый раз, когда метод `read` запрашивает новую входную строку. Ни одна из этих схем не будет оптимальной для всех ситуаций (ввод лучше было бы отобразить на более долгоживущий виджет), но они доказывают правильность общей идеи.

На рис. 10.8 изображена картина, создаваемая реализацией самотестирования этого сценария, после перехвата вывода команды оболочки `dir` в Windows (слева) и двух интерактивных проверок цикла (окно с приглашениями «Line?» и прописными буквами представляет работу теста `makeUpper` перенаправления потоков). Диалог ввода выведен для демонстрации нового теста интерфейса файлов `makeLower`.

Возможно, эта картина не настолько захватывающая, чтобы на нее можно было смотреть часами, но она отражает автоматическое отображение файловых операций ввода-вывода в виджеты графического интерфейса – как будет показано чуть ниже, это решает большую часть последней проблемы, обозначенной в предыдущем разделе.

Но прежде чем двинуться дальше, необходимо отметить, что реализованный в этом модуле вызов функции, для которой выполняется перенаправление потоков ввода-вывода, а также его цикл чтения вывода по-

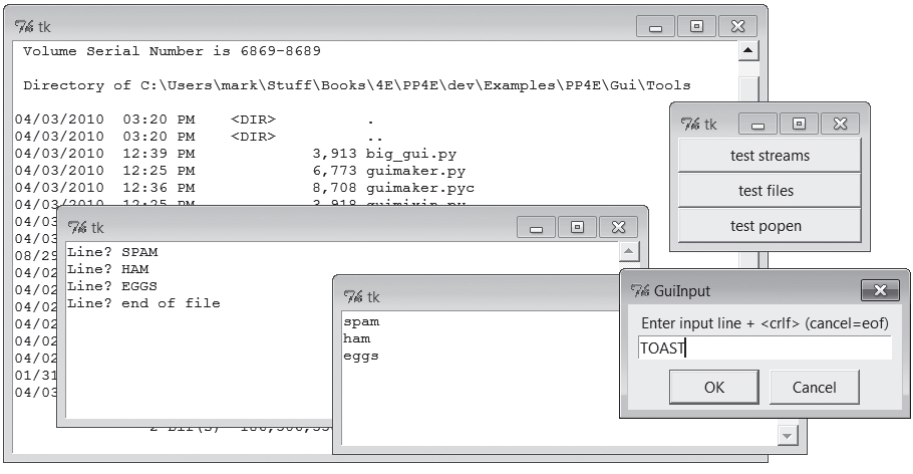


Рис. 10.8. *Сценарий guiStreams направляет потоки во всплывающие окна*

рожденной команды оболочки, могут быть *заблокированы* – они не возвращают управление циклу событий графического интерфейса, пока не завершится функция или запущенная команда оболочки. И хотя класс `GuiOutput` предусматривает вызов метода `update` из библиотеки tkinter после записи каждой строки, тем не менее в целом этот модуль не имеет возможности контролировать продолжительность выполнения функций или команд, которые он запускает.

В функции `redirectedGuiShellCmd`, например, вызов метода `input.readline` будет вызывать задержку, пока от порожденной программы не будет принята полная выходная строка, и графический интерфейс в течение этого времени не будет откликаться на действия пользователя. Поскольку объект вывода `output` вызывает метод `update`, изображение на экране будет обновляться в процессе выполнения программы (метод `update` немедленно вызывает цикл событий Tk), но не чаще, чем будут приниматься строки от порожденной программы. Кроме того, из-за наличия цикла в этой функции графический интерфейс полностью подчиняет себя запущенной им команде, пока она не завершится.

Вызовы функций в `redirectedGuiFunc` также подвержены подобным блокировкам. Кроме того, на протяжении всего времени работы вызываемой функции графический интерфейс обновляется не чаще, чем функция будет выводить данные. Иными словами, эта упрощенная блокирующая модель может стать источником проблем в крупных графических интерфейсах. Мы еще вернемся к этой теме далее, когда встретимся с потоками выполнения. А пока данная реализация вполне соответствует нашим текущим целям.

Использование перенаправления для сценариев архивирования

Теперь, чтобы использовать эти инструменты перенаправления для отображения вывода сценария командной строки в графический интерфейс, просто выполним вызовы и команды оболочки через две функции этого модуля. Пример 10.13 демонстрирует один из способов обертывания вызова диалога архивирования, реализация которого представлена в примере 10.10, благодаря которому вывод операции оказывается во всплывающем окне вместо консоли.

Пример 10.13. PP4E\Gui\ShellGui\packdlg-redirect.py

```
# обертывает запуск сценария командной строки инструментом перенаправления его
# вывода в графический интерфейс

from tkinter import *
from packdlg import runPackDialog
from PP4E.Gui.Tools.guiStreams import redirectedGuiFunc

def runPackDialog_Wrapped():          # обработчик для использования в
    redirectedGuiFunc(runPackDialog) # модуле mytools.py, обертывает прежний
                                      # обработчик целиком
if __name__ == '__main__':
    root = Tk()
    Button(root, text='pop', command=runPackDialog_Wrapped).pack(fill=X)
    root.mainloop()
```

Можете проверить работу этого сценария, запустив его непосредственно, без привлечения окна `ShellGui`. На рис. 10.9 изображено получившееся окно `stdout` после закрытия диалога ввода параметров для операции архивирования. Окно появляется, как только сценарий создаст вывод, и предоставляет пользователю несколько более дружественный графический интерфейс, чем при отлове сообщений в консоли. Аналогичный

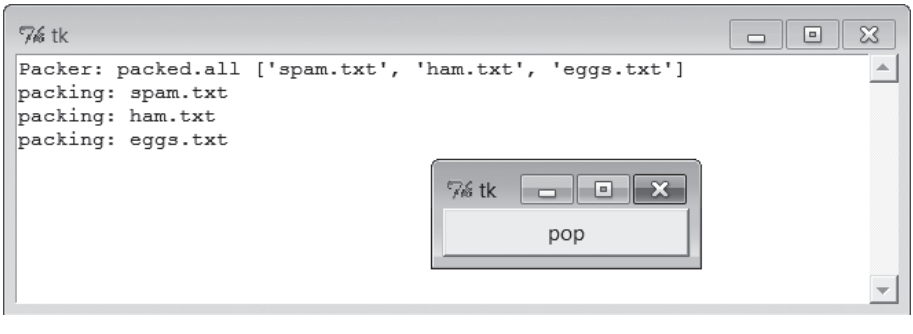


Рис. 10.9. *Перенаправление вывода сценария во всплывающие окна с графическим интерфейсом*

программный код можно написать и для диалога ввода параметров разархивирования, чтобы направить вывод во всплывающее окно. Просто измените сценарий *mytools.py* в примере 10.6, зарегистрировав представленную здесь функцию-обертку в качестве обработчика.

Фактически этот прием можно использовать для перенаправления вывода любой функции или команды оболочки во всплывающее окно. Как обычно, идея совместимых интерфейсов объектов в значительной мере обусловливает гибкость Python.

Динамическая перезагрузка обработчиков

Следующий прием программирования, который мы рассмотрим, касается изменения графического интерфейса в процессе его работы. Функция `imp.reload` в языке Python позволяет динамически изменять и перезагружать 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 перезагрузки измененного модуля.

Это мощная возможность, особенно при разработке программ, перезапуск которых мог бы занять длительное время. Программы, которые подключаются к базам данных или сетевым серверам, инициализируют крупные объекты или проходят длинную последовательность шагов, чтобы снова запустить обработчик, являются первыми кандидатами на использование функции `reload`. Эта функция может существенно сократить время разработки.

Однако в графическом интерфейсе при регистрации обработчиков сохраняются *ссылки на объекты*, а не имена модулей и объектов, поэтому перезагрузка функций обработчиков после их регистрации не даст желаемого эффекта. Операция `imp.reload` действует путем изменения содержимого объекта модуля в памяти. Однако, так как библиотека tkinter запоминает указатель на зарегистрированный объект обработчика, ей неизвестно о перезагрузке модуля, в котором находится обработчик. Это означает, что tkinter по-прежнему будет ссылаться на старые объекты модуля, даже если модуль был изменен и перезагружен.

Это тонкий момент, но в действительности достаточно только запомнить, что для динамической перезагрузки функций обработчиков требуется выполнить особые действия. Необходимо не только явно выполнить перезагрузку измененных модулей, но и предоставить некоторый косвенный слой, маршрутизирующий обратные вызовы от зарегистрированных объектов в модули, чтобы перезагрузка возымела эффект.

Например, сценарий в примере 10.14 выполняет дополнительные действия, перенаправляя обратные вызовы функциям в явно перезагруженном модуле. Обработчики, зарегистрированные в библиотеке tkinter, являются объектами методов, которые всего лишь осуществля-

ют перезагрузку и снова отправляют вызов. Так как доступ к действительным функциям обработчиков происходит через объект модуля, перезагрузка этого модуля приводит к обращению к последним версиям этих функций.

Пример 10.14. PP4E\Gui\Tools\rad.py

```
# перезагружает обработчики динамически

from tkinter import *
import radactions      # получить первоначальные обработчики
from imp import reload # в Python 3.X была перемещена в модуль imp

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(radactions)    # перезагрузить модуль radactions перед вызовом
        radactions.message1() # теперь щелчок на кнопке вызовет новую версию

    def message2(self):
        reload(radactions)         # изменения в radactions.py возымеют эффект
                                   # благодаря перезагрузке
        radactions.message2(self)  # вызовет свежую версию; передать self

    def method1(self):
        print('exposed method...') # вызывается из функции в модуле radactions

Hello().mainloop()
```

Если запустить этот сценарий, он создаст окно с двумя кнопками, вызывающими методы `message1` и `message2`. Пример 10.15 содержит фактическую реализацию обработчика. Его функции получают аргумент `self`, который обеспечивает доступ к объекту класса `Hello`, как если бы это были действительные методы. Можно многократно изменять этот файл во время выполнения сценария `rad`; каждое такое действие изменяет поведение графического интерфейса при нажатии кнопки.

Пример 10.15. PP4E\Gui\Tools\radactions.py

```
# обработчики: перезагружаются перед каждым вызовом
def message1():           # изменить себя
    print('spamSpamSPAM') # можно было бы вывести диалог...

def message2(self):
```

```
print('Ni! Ni!')      # изменить себя
self.method1()        # обращение к экземпляру 'Hello'...
```

Попробуйте запустить сценарий `rad` и изменять сообщения, которые выводит `radactions`, в другом окне. Вы должны увидеть, как при нажатии кнопок в окно консоли будут выводиться новые сообщения. Данный пример намеренно сделан простым, для иллюстрирации идеи, но на практике перезагружаемые таким способом операции могут выводить диалоги, новые окна верхнего уровня и так далее. Перезагрузка программного кода, создающего такие окна, позволяет динамически изменять их внешний вид.

Существуют и другие способы изменения графического интерфейса во время его выполнения. Например, в главе 9 мы видели, что внешний вид в любой момент можно изменить, вызвав метод `config` виджета, а сами виджеты можно динамически добавлять и удалять с экрана такими методами, как `pack_forget` и `pack` (и родственными им для менеджера компоновки `grid`). Кроме того, передача нового значения параметра `command=`*`action`* методу `config` может динамически установить в качестве обработчика обратного вызова новый вызываемый объект – при наличии соответствующей поддержки это может оказаться реальной альтернативой использованной выше обходной схеме повышения эффективности перезагрузки в графических интерфейсах.

Разумеется, далеко не все графические интерфейсы должны быть настолько динамичными. Однако представьте себе игру, которая позволяет модифицировать персонажи, – динамическая перезагрузка в таких системах может оказаться очень полезной. (Я оставляю задачу расширения этого примера на многопользовательские ролевые игры как самостоятельное упражнение.)

Обертывание интерфейсов окон верхнего уровня

Интерфейсы окон верхнего уровня были представлены в главе 8. Данный раздел продолжает обсуждение с того места, где это представление завершилось, и демонстрирует обертывание этих интерфейсов классами, автоматизирующими большую часть операций по созданию окон верхнего уровня, – установку заголовков, поиск и отображение ярлыков окон, выполнение действий по закрытию окна исходя из его назначения, перехват события щелчка мышью на кнопке закрытия окна в его заголовке и так далее.

В примере 10.16 приводятся определения классов-оберток для наиболее часто используемых типов окон – главного окна приложения, временного всплывающего окна и окна встроенного компонента графического интерфейса. Эти типы окон несколько отличаются друг от друга реализациями операции закрытия, но наследуют множество общих черт:

ярлыки, заголовки и кнопки закрытия. Используя непосредственно, подмешивая или наследуя класс окна требуемого типа, вы бесплатно получаете в свое распоряжение все логику настройки.

Пример 10.16. PP4E\Gui\Tools\windows.py

```
"""
################################################################################
Классы, инкапсулирующие интерфейсы верхнего уровня.
Позволяют создавать главные, всплывающие или присоединяемые окна; эти классы
могут наследоваться непосредственно, смешиваться с другими классами или
вызываться непосредственно, без создания подклассов; должны подмешиваться после
(то есть правее) более конкретных прикладных классов: иначе подклассы будут
получать методы (destroy, okayToQuit) из этих, а не из прикладных классов,
и лишатся возможности переопределить их.
################################################################################
"""

import os, glob
from tkinter import Tk, Toplevel, Frame, YES, BOTH, RIDGE
from tkinter.messagebox import showinfo, askyesno

class _window:
    """
    подмешиваемый класс, используется классами главных и всплывающих окон
    """
    foundicon = None           # совместно используется всеми экземплярами
    iconpatt = '*.ico'         # может быть сброшен
    iconmine = 'py.ico'

    def configBorders(self, app, kind, iconfile):
        if not iconfile:               # ярлык не был передан?
            iconfile = self.findIcon() # поиск в тек. каталоге и в каталоге
        title = app                    # модуля
        if kind: title += ' - ' + kind
        self.title(title)              # на рамке окна
        self.iconname(app)             # при свертывании
        if iconfile:
            try:
                self.iconbitmap(iconfile) # изображение ярлыка окна
            except:                       # проблема с интерпретатором или
                pass                      # платформой
        self.protocol('WM_DELETE_WINDOW', self.quit) # не закрывать без
                                                     # подтверждения
    def findIcon(self):
        if _window.foundicon:              # ярлык уже найден?
            return _window.foundicon
        iconfile = None                    # сначала искать в тек. каталоге
        iconshere = glob.glob(self.iconpatt) # допускается только один
        if iconshere:                        # удалить ярлык с красными
            iconfile = iconshere[0]          # буквами Tk
```

```
        else:                                # поиск в каталоге модуля
            mymod = __import__(__name__)     # импортировать, получить каталог
            path = __name__.split('.')       # возможно, путь пакета
            for mod in path[1:]:             # по всему пути до конца
                mymod = getattr(mymod, mod)  # только самый первый
            mydir = os.path.dirname(mymod.__file__)
            myicon = os.path.join(mydir, self.iconmine) # исп. myicon, а не tk
            if os.path.exists(myicon): iconfile = myicon
        _window.foundicon = iconfile         # не выполнять поиск вторично
        return iconfile

class MainWindow(Tk, _window):
    """
    главное окно верхнего уровня
    """
    def __init__(self, app, kind='', iconfile=None):
        Tk.__init__(self)
        self.__app = app
        self.configBorders(app, kind, iconfile)

    def quit(self):
        if self.okayToQuit():      # потоки запущены?
            if askyesno(self.__app, 'Verify Quit Program?'):
                self.destroy()     # завершить приложение
        else:
            showinfo(self.__app, 'Quit not allowed') # или в okayToQuit?

    def destroy(self):         # просто завершить
        Tk.quit(self)          # переопределить, если необходимо

    def okayToQuit(self):      # переопределить, если используются
        return True            # потоки выполнения

class PopupWindow(Toplevel, _window):
    """
    вторичное всплывающее окно
    """
    def __init__(self, app, kind='', iconfile=None):
        Toplevel.__init__(self)
        self.__app = app
        self.configBorders(app, kind, iconfile)

    def quit(self):            # переопределить, если потребуется изменить
        if askyesno(self.__app, 'Verify Quit Window?'): # или вызвать destroy
            self.destroy()                              # чтобы закрыть окно

    def destroy(self):         # просто закрыть окно
        Toplevel.destroy(self) # переопределить, если необходимо

class QuietPopupWindow(PopupWindow):
```

```
    def quit(self):
        self.destroy()         # закрывать без предупреждения

class ComponentWindow(Frame):
    """
    при присоединении к другим интерфейсам
    """
    def __init__(self, parent):        # если не фрейм
        Frame.__init__(self, parent)   # предоставить контейнер
        self.pack(expand=YES, fill=BOTH)
        self.config(relief=RIDGE, border=2)  # перенастроить при необходимости

    def quit(self):
        showinfo('Quit', 'Not supported in attachment mode')

    # destroy из фрейма: просто удалить фрейм # переопределить, если
                                              # необходимо
```

Почему бы просто не определять ярлык приложения и заголовок окна непосредственно вызовом методов? С одной стороны, особенности такого рода нелегко запомнить (в результате вам придется большую часть времени тратить на копирование и вставку кода). С другой стороны, эти классы добавляют высокоуровневые функциональные возможности, реализацию которых иначе пришлось бы добавлять снова и снова. Кроме всего прочего, эти классы обеспечивают автоматический запрос подтверждения завершения и поиск ярлыка. Например, классы окон всего один раз пытаются отыскать файл ярлыка в текущем рабочем каталоге и в каталоге, где находится данный модуль.

Используя классы, инкапсулирующие – то есть скрывающие – такие подробности, мы получаем всю мощь инструмента, без необходимости задумываться об их реализации в будущем. Кроме того, с помощью этих классов мы можем придавать нашим приложениям стандартизованный внешний вид и поведение. А если в будущем их потребуется изменить, нам достаточно будет изменить программный код только в одном месте, а не во всех реализованных нами окнах.

Для тестирования этого модуля в примере 10.17 приводится сценарий, использующий эти классы в различных режимах – в качестве подмешиваемых классов, в качестве суперклассов и непосредственно – из обычного процедурного программного кода.

Пример 10.17. PP4E\Gui\Tools\windows-test.py

```
# модуль windows должен импортироваться, иначе атрибут __name__ будет иметь
# значение __main__ в функции findIcon

from tkinter import Button, mainloop
from windows import MainWindow, PopupWindow, ComponentWindow

def _selftest():
```

```
    # использовать, как подмешиваемый класс
    class content:
        "используется так же, как Tk, Toplevel и Frame"
        def __init__(self):
            Button(self, text='Larch', command=self.quit).pack()
            Button(self, text='Sing ', command=self.destroy).pack()

    class contentmix(MainWindow, content):
        def __init__(self):
            MainWindow.__init__(self, 'mixin', 'Main')
            content.__init__(self)
    contentmix()

    class contentmix(PopupWindow, content):
        def __init__(self):
            PopupWindow.__init__(self, 'mixin', 'Popup')
            content.__init__(self)
    prev = contentmix()

    class contentmix(ComponentWindow, content):
        def __init__(self):                       # вложенный фрейм
            ComponentWindow.__init__(self, prev) # в предыдущем окне
            content.__init__(self)                # кнопка Sing стирает фрейм
    contentmix()

    # использовать в подклассах
    class contentsub(PopupWindow):
        def __init__(self):
            PopupWindow.__init__(self, 'popup', 'subclass')
            Button(self, text='Pine', command=self.quit).pack()
            Button(self, text='Sing', command=self.destroy).pack()
    contentsub()

    # использование в процедурном программном коде
    win = PopupWindow('popup', 'attachment')
    Button(win, text='Redwood', command=win.quit).pack()
    Button(win, text='Sing   ', command=win.destroy).pack()
    mainloop()

if __name__ == '__main__':
    _selftest()
```

Если запустить этот тест, он создаст четыре окна, как показано на рис. 10.10. Все окна автоматически получат ярлык с голубыми буквами «PY» и будут перехватывать и запрашивать подтверждение при попытке закрыть их щелчком на кнопке X в правом верхнем углу, благодаря логике поиска и настройки, унаследованной из классов окон в модуле. Некоторые кнопки в окнах, воспроизводимых тестовым сценарием, закрывают только вмещающее их окно, некоторые – все приложение, некоторые стирают только присоединенное окно, а некоторые выводят

диалог с просьбой подтвердить закрытие окна. Запустите этот сценарий у себя на компьютере, чтобы увидеть, как действуют различные кнопки, и получить возможность сопоставить их поведение с реализацией в сценарии – действия, выполняемые при закрытии окна, зависят от его типа.

Рис. 10.10. *Интерфейс сценария windows-test*

Мы будем использовать эти классы-обертки в следующей главе, в примере PyClock, и еще раз – в главе 14, где они будут использоваться, чтобы уменьшить сложность программы PyMailGUI. Отчасти преимущество применения приемов ООП в языке Python состоит в том, что благодаря им мы можем позднее не вспоминать детали реализации.

Графические интерфейсы, потоки выполнения и очереди

В главе 5 мы познакомились с потоками выполнения и механизмом очередей, который обычно используется для организации обмена данными между потоками. Там же было дано краткое описание применения этих идей в приложениях с графическим интерфейсом. В главе 9 мы продолжили развитие этих тем применительно к библиотеке tkinter, используемой в этой книге, и расширили модель многопоточных графических интерфейсов в целом, рассмотрев поддержку многопоточной модели выполнения (или ее отсутствие) и назначение очередей и блокировок.

Теперь, когда мы приобрели определенный опыт разработки графических интерфейсов, мы можем, наконец, перейти к воплощению этих идей в программный код. Если в свое время вы пропустили описание этих тем в главе 5 или 9, вам, вероятно, лучше вернуться назад и прочитать их – мы не будем повторно рассматривать здесь основы программирования многопоточных приложений или применения очередей.

Многопоточная модель имеет самое прямое отношение к графическим интерфейсам. Напомню, что продолжительные операции вообще должны выполняться в параллельных потоках, чтобы избежать блокирования графического интерфейса и обеспечить его отзывчивость на действия пользователя. Под продолжительными операциями обычно понимаются вызовы функций, которые выполняются значительное время, операции загрузки данных с серверов, блокирующие операции ввода-вывода и любые другие, которые могут вызвать заметную задержку. В обсуждении нашего примера со сценариями архивирования/разархивирования, представленного выше в этой главе, например, отмечалось, что функции, выполняющие фактическую обработку файлов, в целом должны выполняться в отдельных потоках, чтобы не блокировать главный поток графического интерфейса до их завершения.

В общем случае, когда графический интерфейс ожидает завершения какой-либо операции, он становится полностью неотзывчивым – окно не сможет изменить размер, свернуться и даже просто перерисовать себя, если он был перекрыт другим окном, которое потом ушло. Чтобы исключить возможность такого блокирования, программы с графическим интерфейсом должны выполнять продолжительные операции параллельно, обычно с применением механизма потоков выполнения, который позволяет совместно использовать данные программы. При таком подходе главный поток выполнения графического интерфейса получает возможность обновлять изображение на экране и откликаться на действия пользователя, в то время как другие потоки заняты решением других задач. Как мы уже видели, в некоторых ситуациях на помощь может прийти метод `update`, но его можно считать решением проблем, только когда есть возможность вызвать его, – потоки выполнения обеспечивают по-настоящему параллельное выполнение продолжительных операций и предлагают более универсальное решение.

Однако, как мы узнали в главе 9, только главный поток выполнения должен обновлять графический интерфейс – потоки выполнения, выполняющие продолжительные операции, не должны делать этого. Вместо этого они должны помещать данные в очередь (или передавать их через иной механизм), чтобы главный поток мог их извлечь и отобразить. Для этого в главном потоке обычно выполняется цикл опроса через определенные интервалы времени, в котором проверяется поступление новых результатов для отображения. Дочерние потоки производят данные и помещают их в очередь, но они ничего не знают о графическом интерфейсе, – главный поток получает данные и отображает их, но не занимается их генерированием.

Вследствие такого разделения труда мы обычно называем эту модель *производитель/потребитель* – вычислительные потоки производят данные, которые потребляются главным потоком графического интерфейса. Потоки, выполняющие продолжительные операции, иногда называют *рабочими* потоками, потому что они выполняют работу по производству результатов, которые затем представляются пользовате-

лю с помощью графического интерфейса. В некотором смысле, графический интерфейс является клиентом рабочих потоков-серверов, хотя обычно эта терминология используется в более узком смысле – серверы являются долгоживущими источниками данных, не имеющими тесной связи с клиентами (хотя графический интерфейс также может отображать данные, полученные от независимых серверов). Но, независимо от своего названия, данная модель позволяет исключить блокирование графического интерфейса и параллельно выполнять другие задачи, которые не выполняют непосредственного обновления графического интерфейса.

В качестве конкретного примера представим, что наш графический интерфейс должен отображать данные телеметрии, получаемые со спутника в масштабе реального времени через сокеты (механизм взаимодействий между процессами, представленный в главе 5). Такая программа должна быть достаточно отзывчивой, чтобы не потерять входящие данные, и при этом не должна блокироваться в периоды ожидания или обработки данных. Чтобы достичь обеих целей, можно породить дочерние потоки выполнения, которые будут получать поступающие данные и помещать их в очередь, а главный поток графического интерфейса будет периодически извлекать данные из очереди и отображать их. При таком разделении труда графический интерфейс не блокируется данными со спутника и наоборот – сам графический интерфейс будет выполняться независимо от потоков данных, а поскольку потоки обработки данных могут выполняться на полной скорости, они смогут принимать входные данные с той же скоростью, с какой они будут отправляться. Вообще, циклы событий графических интерфейсов не настолько отзывчивы, чтобы обрабатывать поступающие данные в масштабе реального времени. Без дополнительных потоков выполнения мы могли бы потерять часть телеметрии, а с ними мы сумеем принимать все отправляемые данные и отображать их, как только цикл событий графического интерфейса найдет время, чтобы извлечь их из очереди, – достаточно быстро, чтобы не вызвать ощущения задержки у пользователя. В отсутствие данных только дочерние потоки будут ожидать их появления, но не графический интерфейс.

В других ситуациях дополнительные потоки могут потребоваться, только чтобы графический интерфейс оставался активным в ходе выполнения продолжительных операций. Загружая данные с веб-сервера, например, графический интерфейс должен иметь возможность перерисовывать себя при перекрытии другими окнами или при изменении размеров. По этой причине вызов функции загрузки не может быть простым вызовом функции – функция должна выполняться параллельно остальной программе, обычно в отдельном потоке. По окончании загрузки поток должен известить графический интерфейс, что данные готовы для отображения, поместив их в очередь, – главный поток обнаружит их при следующей же проверке очереди в обработчике, вызываемом по

таймеру. Например, мы будем использовать потоки и очереди таким способом в программе PyMailGUI, в главе 14, чтобы обеспечить возможность параллельного выполнения нескольких операций передачи почты без блокирования графического интерфейса.

Помещение данных в очередь

Обменивается ли ваш графический интерфейс данными со спутниками, веб-серверами или с чем-то еще, эта многопоточная модель легко воплощается в программный код. В примере 10.18 приводится графический интерфейс, эквивалентный многопоточной программе с очередями, с которой мы встречались в главе 5 (сравните его с примером 5.14). В данном случае графический интерфейс является потоком-потребителем, а потоки-производители добавляют данные для отображения в общую очередь. Для проверки появления результатов в очереди главный поток вместо явного цикла использует метод `after` из библиотеки.

Пример 10.18. PP4E\Gui\Tools\queuetest-gui.py

```
# графический интерфейс, отображающий данные, производимые рабочими потоками

import _thread, queue, time
dataQueue = queue.Queue()      # бесконечной длины

def producer(id):
    for i in range(5):
        time.sleep(0.1)
        print('put')
        dataQueue.put('[producer id=%d, count=%d]' % (id, i))

def consumer(root):
    try:
        print('get')
        data = dataQueue.get(block=False)
    except queue.Empty:
        pass
    else:
        root.insert('end', 'consumer got => %s\n' % str(data))
        root.see('end')
    root.after(250, lambda: consumer(root))    # 4 раза в секунду

def makethreads():
    for i in range(4):
        _thread.start_new_thread(producer, (i,))

if __name__ == '__main__':
    # главный поток: порождает группу рабочих потоков на каждый щелчок мыши
    from tkinter.scrolledtext import ScrolledText
    root = ScrolledText()
    root.pack()
```

```
root.bind('<Button-1>', lambda event: makethreads())
consumer(root)     # запустить цикл проверки очереди в главном потоке окна
root.mainloop()    # вход в цикл событий
```

Обратите внимание, что здесь при каждом событии от таймера из очереди извлекается только один элемент данных. Это было сделано сознательно. Можно было бы при каждом событии от таймера извлекать все элементы из очереди в цикле. Но в критических случаях, при наличии большого количества данных в очереди, это легко могло бы привести к блокированию графического интерфейса (представьте быстрый интерфейс телеметрии, который внезапно передал в очередь сразу сотни или даже тысячи результатов измерений). Обрабатывая по одному элементу за раз, мы гарантируем, что цикл событий графического интерфейса будет получать управление, обновлять изображение на экране и обрабатывать ввод пользователя. Недостаток такого подхода состоит в том, что при большом количестве элементов данных в очереди их обработка может занять продолжительное время. В подобных ситуациях можно воспользоваться гибридными схемами, например извлекать из очереди до N элементов данных при каждом событии от таймера, – мы увидим пример реализации такой схемы далее в этом разделе (пример 10.20).

Если запустить этот сценарий, главный поток графического интерфейса начнет извлекать данные из очереди и отображать их в окне `ScrolledText`, как показано на рис. 10.11. При каждом щелчке левой кнопкой мыши в окне будет запускаться новая группа из четырех потоков-производителей. Потоки выполнения выводят сообщения «get» и «put» в стандартный поток вывода (в данном примере эта операция не синхронизируется между потоками – на некоторых платформах, включая Windows, выводимые сообщения могут перемешиваться). Для имитации выполнения продолжительных операций, таких как получение почты, извлечение результатов запроса или ожидание поступления данных через сокет (дополнительно о сокетах рассказывается ниже в этой главе), потоки-производители вызывают функцию `sleep`. Я выполнил несколько щелчков левой кнопкой мыши, чтобы обеспечить перекрытие потоков выполнения, как показано на рис. 10.11.

Реализация с классами и связанными методами

Пример 10.19 представляет дальнейшее развитие этой модели, реализовав ее в виде класса, обеспечив тем самым возможность специализации и повторного использования. По своему действию и внешнему виду окна этот пример ничем не отличается от процедурной версии, но очередь в нем проверяется чаще и ничего не выводится в стандартный поток вывода. Обратите внимание, что в качестве обработчика событий от мыши и функции потока выполнения здесь используются связанные методы – связанные методы хранят в себе ссылку на экземпляр и на сам метод, поэтому при выполнении в потоке он обладает возможно-

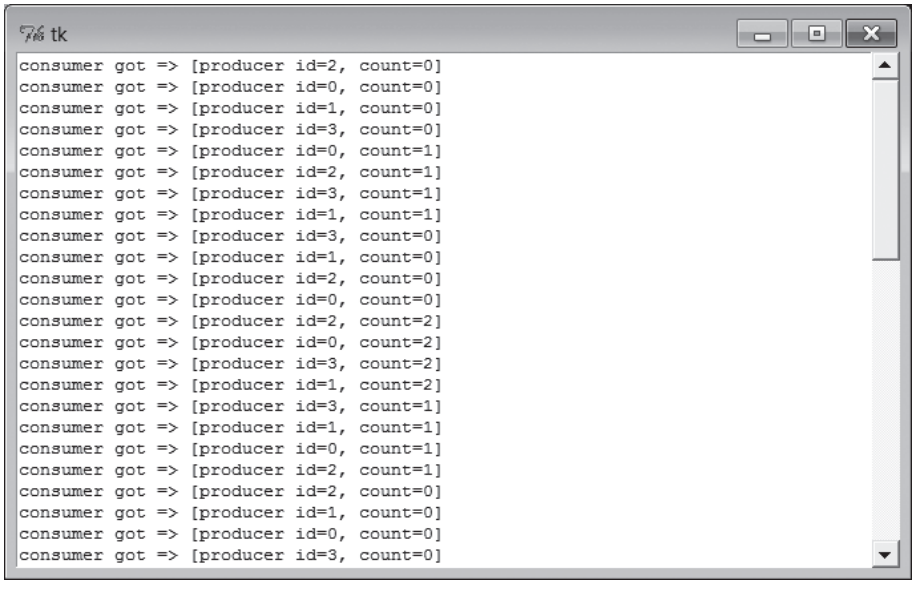


Рис. 10.11. *Изображение обновляется главным потоком графического интерфейса*

стью доступа к информации в объекте, включая очередь. Это позволило перенести очередь и окно из глобальных переменных в атрибуты экземпляра.

Пример 10.19. PP4E\Gui\Tools\queuetest-gui-class.py

```
# графический интерфейс, отображающий данные, производимые рабочими потоками
# (на основе классов)

import threading, queue, time
from tkinter.scrolledtext import ScrolledText   # или PP4E.Gui.Tour.scrolledtext

class ThreadGui(ScrolledText):
    threadsPerClick = 4

    def __init__(self, parent=None):
        ScrolledText.__init__(self, parent)
        self.pack()
        self.dataQueue = queue.Queue()              # бесконечной длины
        self.bind('<Button-1>', self.makethreads)  # по щелчку левой кнопкой
        self.consumer()                             # цикл проверки очереди в
                                                    # главном потоке выполнения

    def producer(self, id):
        for i in range(5):
            time.sleep(0.1)
```

```
            self.dataQueue.put('[producer id=%d, count=%d]' % (id, i))

    def consumer(self):
        try:
            data = self.dataQueue.get(block=False)
        except queue.Empty:
            pass
        else:
            self.insert('end', 'consumer got => %s\n' % str(data))
            self.see('end')
        self.after(100, self.consumer)             # 10 раз в секунду

    def makethreads(self, event):
        for i in range(self.threadsPerClick):
            threading.Thread(target=self.producer, args=(i,)).start()

if __name__ == '__main__':
    root = ThreadGui() # в главном потоке: создать GUI, запустить цикл таймера
    root.mainloop()    # войти в цикл событий tk
```

Рассмотрите внимательнее организацию потоков, использование общей очереди и извлечение данных в цикле от таймера, так как с этими приемами мы еще встретимся далее в этой главе, а также в главе 11, в примере программы PyEdit. В программе PyEdit мы будем использовать эти приемы для организации поиска внешних файлов в потоках, чтобы избежать блокирования графического интерфейса и обеспечить возможность одновременного поиска сразу нескольких файлов. Кроме того, мы повторно воспользуемся классической многопоточной моделью производитель/потребитель в более реалистичном примере, представленном далее в этой главе. позволяющей избежать блокирования графического интерфейса, который должен читать данные из стандартного потока ввода, связанного с потоком вывода другой программы.

Завершение потоков в графических интерфейсах

Кроме всего прочего, в примере 10.19 вместо модуля `_thread` используется модуль `threading`. Это означает, что в отличие от предыдущей версии, программа не завершится, пока выполняются какие-либо потоки-производители, если только они не были запущены как потоки-демоны, установкой их флагов `daemon` в значение `True`. Напомню, что при использовании модуля `threading` программы завершаются, только когда остаются одни потоки-демоны, – потоки-производители наследуют значение `False` в атрибуте `daemon` от потока, создавшего их, что препятствует завершению программы, пока они продолжают выполняться.

Однако в данном примере дочерние потоки выполнения завершаются слишком быстро, чтобы можно было заметить задержку при завершении программы. Измените в сценарии вызов функции `time.sleep`, чтобы он выполнял задержку на 2 секунды, имитируя долгоживущий рабочий поток, и попробуйте запустить пример. Попытка закрыть окно

после щелчка левой кнопкой мыши сотрет окно, но сама программа продолжит выполняться еще в течение примерно 10 секунд (это можно наблюдать, например, в виде паузы в окне консоли). Если сделать то же самое в предыдущей версии, использующей модуль `_thread`, или в этой версии установить флаги `daemon` потоков в значение `True`, программа будет завершаться немедленно.

При решении практических задач может потребоваться взвесить различные политики управления завершением в контексте действующих потоков и программировать их соответствующим образом; при необходимости отложить завершение программы можно использовать потоки, запущенные со значением `False` в атрибуте `daemon`, или блокировки. Напротив, использование потоков `threading` может препятствовать желательному завершению программы, если забыть установить флаг `daemon` в значение `True`. Дополнительно о завершении программ и о потоках-демонах (и о других пугающих темах!) рассказывается в главе 5.

Помещение обработчиков в очередь

В примерах предыдущего раздела данные, помещаемые в очередь, всегда были строками. Этого вполне достаточно для простых приложений, где могут существовать производители только одного типа. Однако если в программе могут иметься потоки, выполняющие различные функции и производящие данные различных типов, это может осложнить обработку данных. Вероятно, вам придется сопровождать данные некоторой дополнительной информацией, которая поможет главному потоку графического интерфейса определить, как обрабатывать эти данные.

Представьте, например, почтовый клиент, где несколько операций отправки и приема почты могут выполняться одновременно. Если все потоки совместно используют одну и ту же очередь, информация, помещаемая в нее, должна иметь какие-то отличительные признаки, которые позволят определить, какое событие она представляет, – загруженное сообщение для отображения, информацию для индикатора хода выполнения операции, сообщение об успешной отправке или что-то еще. Это надуманный пример: мы столкнемся с этой проблемой в приложении PyMailGUI, представленном в главе 14.

К счастью, очереди поддерживают не только строки – в очередь можно помещать объекты Python любых типов. Наиболее универсальными из них, вероятно, являются вызываемые объекты: помещая в очередь функцию или другой вызываемый объект, поток-производитель может самым непосредственным способом сообщить графическому интерфейсу, как обрабатывать данные. Графическому интерфейсу остается просто извлечь объект из очереди и вызвать его. Поскольку все потоки выполняются в пределах одного и того же процесса, в очередь могут помещаться вызываемые объекты любых типов – простые функции, результаты `lambda`-выражений и даже связанные методы, объединяющие в себе функции и объекты, обеспечивающие доступ к их данным и ме-

тодам. Любые изменения в объектах, выполняемые такими функциями обратного вызова, будут доступны всему процессу.

Благодаря возможности в языке Python универсальным способом обрабатывать функции и списки их аргументов, передача их через очередь выглядит гораздо проще, чем могло бы показаться. Так, в примере 10.20 демонстрируется один из способов передачи функций обратного вызова через очередь, который будет использоваться в приложении PyMailGUI в главе 14. Этот модуль содержит также ряд полезных инструментов. Класс `ThreadCounter` можно использовать как совместно используемый счетчик и логический флаг (например, для управления операциями, перекрывающимися во времени). Однако наиболее важным здесь является реализация интерфейса передачи функций через очередь – в двух словах, данная реализация позволяет клиентам запускать потоки выполнения, которые помещают в очередь свои функции завершения для передачи главному потоку.

До определенной степени этот пример можно считать лишь разновидностью примера из предыдущего раздела – здесь мы по-прежнему выполняем цикл событий от таймера, в котором главный поток извлекает данные из очереди. Для большей эффективности за одно событие от таймера из очереди извлекается уже не один (что может оказаться слишком долгим, при большом количестве элементов данных, или слишком накладным, при коротком интервале срабатывания таймера) и не все (что может привести к блокированию графического интерфейса, если данные поступают слишком быстро), а до N элементов данных. Мы дополним этот прием пакетной обработки данных в PyMailGUI возможностью выполнения множественных изменений в графическом интерфейсе без необходимости выделять ресурсы процессора для обработки коротких событий от таймера, в чем обычно нет необходимости.

Однако здесь главное отличие, на которое следует обратить внимание, заключается в том, что мы *вызываем* объекты, которые потоки-производители обобщенным способом помещают в очередь для обработки успешного или неудачного выполнения операции в ответ на благополучное завершение или возникшее исключение. Кроме того, функции, выполняемые в потоках-производителях, принимают функцию, определяющую информацию о протекании операции, которая при вызове просто помещает в очередь обработчик индикатора хода выполнения операции, предназначенный для вызова в контексте главного потока выполнения. Эту особенность можно использовать, например, чтобы показать в графическом интерфейсе ход выполнения загрузки данных по сети.

Пример 10.20. PP4E\Gui\Tools\threadtools.py

```
"""
###############################################################################
Общесистемные утилиты поддержки многопоточной модели выполнения для графических
интерфейсов.
```

```
Реализует единую очередь обработчиков и цикл обработки событий от таймера для
ее проверки, совместно используемые всеми окнами в программе; рабочие потоки
помещают в очередь свои обработчики завершения и протекания операции для вызова
в главном потоке; эта модель не блокирует графический интерфейс – он просто
выполняет операции в порождаемых дочерних потоках и обрабатывает события
завершения и продолжения операций; рабочие потоки могут перекрываться во времени
с главным потоком и с другими рабочими потоками.

На практике передача функций-обработчиков с аргументами через очереди намного
удобнее, чем передача простых данных, если в программе одновременно могут
действовать разнотипные потоки выполнения, – каждый тип может подразумевать
выполнение различных действий при завершении.

Библиотеки создания графических интерфейсов не полностью поддерживают
многопоточную модель, поэтому, вместо того чтобы напрямую вызывать обработчики,
производящие изменение графического интерфейса после выполнения основной
операции в потоке, они помещаются в общую очередь и вызываются не в дочерних
потоках, а в цикле обработки событий от таймера в главном потоке; это также
обеспечивает регулярность и предсказуемость моментов обновления графического
интерфейса; требуется, чтобы логика потока разбивалась на основную операцию,
завершающие действия и операцию, возвращающую информацию о протекании процесса.

Предполагается, что в случае неудачи функция потока возбуждает исключение
и принимает в аргументе 'progress' функцию обратного вызова, если поддерживает
возможность передачи информации о ходе выполнения операции; предполагается
также, что все обработчики выполняются очень быстро, либо производят обновление
графического интерфейса в процессе работы, и эта очередь будет содержать функции
обратного вызова (или другие вызываемые объекты) для использования в приложениях
с графическим интерфейсом, – требуется наличие виджетов, чтобы обеспечить работу
цикла на основе метода 'after'; для использования данной модели в сценариях без
графического интерфейса можно было бы использовать простой таймер.
################################################################################
"""

# запустить, даже если нет потоков # сейчас, если модуль threads
try:                               # недоступен в стандартной библиотеке,
    import _thread as thread       # возбуждает исключение ImportError
except ImportError:                # и блокирует графический интерфейс
    import _dummy_thread as thread # тот же интерфейс без потоков

# общая очередь
# в глобальной области видимости, совместно используется потоками
import queue, sys
threadQueue = queue.Queue(maxsize=0) # infinite size

################################################################################
# ГЛАВНЫЙ ПОТОК – периодически проверяет очередь; выполняет действия,
# помещаемые в очередь, в контексте главного потока; один потребитель (GUI) и
# множество производителей (загрузка, удаление, отправка); простого списка
# было бы вполне достаточно, если бы операции list.append и list.pop были
```

```
# атомарными; 4 издание: в процессе обработки каждого события от таймера
# выполняет до N операций: обход в цикле всех обработчиков, помещенных в
# очередь, может заблокировать графический интерфейс, а при выполнении
# единственной операции вызов всех обработчиков может занять продолжительное
# время или привести к неэффективному расходованию ресурсов процессора на
# обработку событий от таймера (например, информирование о ходе выполнения
# операций); предполагается, что обработчики выполняются очень быстро или
# выполняют обновление графического интерфейса в процессе работы (вызывают
# метод update): после вызова обработчика планируется очередное событие от
# таймера и управление возвращается в цикл событий; поскольку этот цикл
# выполняется в главном потоке, он не препятствует завершению программы;
################################################################################

def threadChecker(widget, delayMsecs=100, perEvent=1): # 10 раз/сек, 1/таймер
    for i in range(perEvent):                    # передайте другие значения,
        try:                                     # чтобы повысить скорость
            (callback, args) = threadQueue.get(block=False) # выполнить до N
        except queue.Empty:                                 # обработчиков
            break                                           # очередь пуста?
        else:
            callback(*args)                                 # вызвать обраб.

    widget.after(delayMsecs,                                # переустановить
        lambda: threadChecker(widget, delayMsecs, perEvent))# таймер  и
                                                            # вернуться в цикл
                                                            # событий
################################################################################
# НОВЫЙ ПОТОК - выполняет задание, помещает в очередь обработчик завершения и
# обработчик, возвращающий информацию о протекании процесса; вызывает функцию
# основной операции с аргументами, затем планирует вызов функций on* с
# контекстом; запланированные вызовы добавляются в очередь и выполняются в
# главном потоке, чтобы избежать параллельного обновления графического
# интерфейса; позволяет программировать основные операции вообще без учета
# того, что они будут выполняться в потоках; не вызывайте обработчики в
# потоках: они могут обновлять графический интерфейс в потоке, поскольку
# передаваемая функция будет вызвана в потоке; обработчик 'progress' просто
# должен добавлять в очередь функцию обратного вызова с передаваемыми ей
# аргументами; не обновляйте текущие счетчики здесь: обработчик завершения
# будет извлечен из очереди и выполнен функцией threadChecker в главном
# потоке;
################################################################################

def threaded(action, args, context, onExit, onFail, onProgress):
    try:
        if not onProgress: # ждать завершения этого потока
            action(*args)  # предполагается, что в случае неудачи будет
        else:              # возбуждено исключение
            def progress(*any):
                threadQueue.put((onProgress, any + context))
            action(progress=progress, *args)
    except:
```

```
        threadQueue.put((onFail, (sys.exc_info(), ) + context))
    else:
        threadQueue.put((onExit, context))

def startThread(action, args, context, onExit, onFail, onProgress=None):
    thread.start_new_thread(
        threaded, (action, args, context, onExit, onFail, onProgress))

##############################################################################
# счетчик или флаг с поддержкой многопоточной модели выполнения: удобно
# использовать, чтобы избежать выполнения перекрывающихся во времени операций,
# когда потоки изменяют другие общие данные, помимо тех, которые изменяются
# обработчиками, помещаемыми в очередь
##############################################################################

class ThreadCounter:
    def __init__(self):
        self.count = 0
        self.mutex = thread.allocate_lock() # или Threading.semaphore
    def incr(self):
        self.mutex.acquire()                # или с помощью self.mutex:
        self.count += 1
        self.mutex.release()
    def decr(self):
        self.mutex.acquire()
        self.count -= 1
        self.mutex.release()
    def __len__(self): return self.count    # True/False, если используется,
                                            # как флаг
##############################################################################
# реализация самотестирования: разбивает поток на основную операцию,
# операцию завершения и операцию информирования о ходе выполнения задания
##############################################################################

if __name__ == '__main__':     # самотестирование при запуске в виде сценария
    import time                # или PP4E.Gui.Tour.scrolledtext
    from tkinter.scrolledtext import ScrolledText

    def onEvent(i):            # реализация порождения потоков
        myname = 'thread-%s' % i
        startThread(
            action     = threadaction,
            args       = (i, 3),
            context    = (myname,),
            onExit     = threadexit,
            onFail     = threadfail,
            onProgress = threadprogress)

    # основная операция, выполняемая потоком
    def threadaction(id, reps, progress):  # то, что делает поток
        for i in range(reps):
```

```
            time.sleep(1)
            if progress: progress(i)       # обработчик progress: в очередь
        if id % 2 == 1: raise Exception    # ошибочный номер: неудача

    # обработчики завершения/информирования о ходе выполнения задания:
    # передаются главному потоку через очередь
    def threadexit(myname):
        text.insert('end', '%s\texit\n' % myname)
        text.see('end')

    def threadfail(exc_info, myname):
        text.insert('end', '%s\tfail\t%s\n' % (myname, exc_info[0]))
        text.see('end')

    def threadprogress(count, myname):
        text.insert('end', '%s\tprog\t%s\n' % (myname, count))
        text.see('end')
        text.update()  # допустимо: выполняется в главном потоке

    # создать графический интерфейс и запустить цикл обработки событий от
    # таймера в главном потоке
    # порождать группу рабочих потоков в ответ на каждый щелчок мышью:
    # выполнение их может перекрываться во времени

    text = ScrolledText()
    text.pack()
    threadChecker(text)        # запустить цикл обработки потоков
    text.bind('<Button-1>',    # в 3.x функция list необходима для получения
        lambda event: list(map(onEvent, range(6))) ) # всех результатов map,
                                                     # для range - нет
    text.mainloop()            # вход в цикл событий
```

Реализация модуля описывается в комментариях, а программный код самотестирования демонстрирует порядок его использования. Обратите внимание, что реализация потока выполнения разбита на основные операции, операции, выполняемые при выходе, и необязательные операции информирования о ходе выполнения задания, – основные операции выполняются в новом потоке, но другие помещаются в очередь и выполняются в главном потоке. То есть чтобы воспользоваться этим модулем, вам, по сути, необходимо разделить реализацию функции потока на действия, выполняемые в самом потоке, выполняемые после его завершения и необязательные действия информирования о ходе выполнения задания. В целом, продолжительным может быть только этап, выполняемый в пределах потока.

Если пример 10.20 запустить, как самостоятельный сценарий, по каждому щелчку мышью на виджете `ScrolledTest` он будет запускать шесть новых потоков выполнения, каждый из которых будет выполнять функцию `threadaction`. Функция, выполняемая в потоке, вызывает передаваемую ей функцию информирования о ходе выполнения задания,

помещающую обработчик обратного вызова в очередь, который вызывает функцию `threadprogress` в главном потоке. Когда функция потока завершается, интерфейсный уровень помещает в очередь обработчик, который вызовет `threadfail` или `threadexit` в главном потоке, в зависимости от того, возбудила функция потока исключение или нет. Так как все обработчики, помещаемые в очередь, извлекаются и выполняются в цикле обработки событий от таймера в главном потоке, это означает, что все изменения в графическом интерфейсе будут производиться только в главном потоке выполнения и не будут перекрываться во времени.

На рис. 10.12 приводится фрагмент вывода, сгенерированного щелчком мыши на окне. Все сообщения о завершении, неудаче и информирующие о ходе выполнения задания, создаются обработчиками, добавляемыми в очередь дочерними потоками и вызываемыми в цикле обработки событий от таймера в главном потоке.

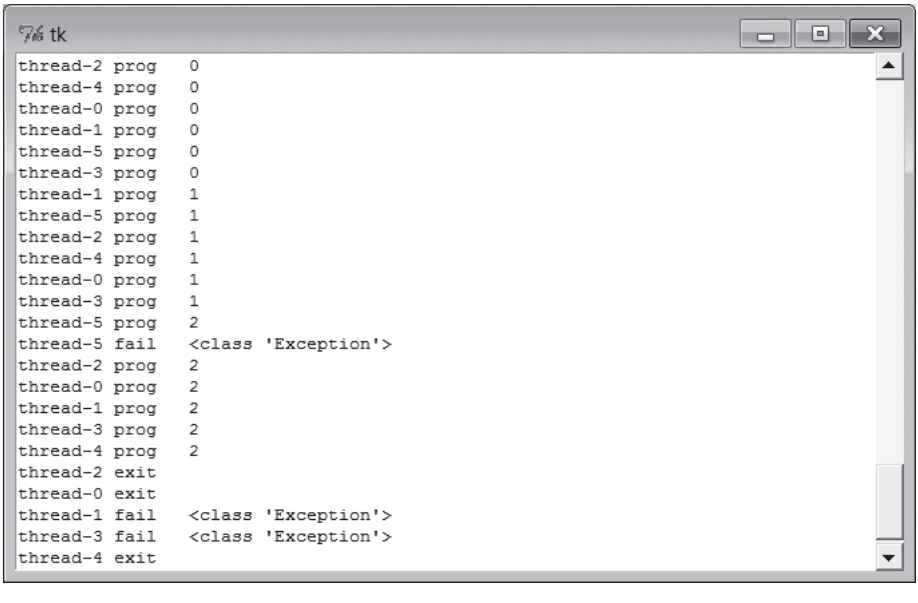


Рис. 10.12. *Сообщения, создаваемые обработчиками, извлекаемыми из очереди*

Внимательно рассмотрите этот пример и попробуйте последить логику работы программного кода самотестирования. Это достаточно сложное задание, и вам, вероятно, придется не один раз пробежать по нему глазами, чтобы понять, как он действует. Однако как только вы уловите суть этой парадигмы, вы поймете общую схему работы с разнородными потоками выполнения обобщенным способом. В PyMailGUI, например, когда требуется запустить прием/передачу почты, выполняются операции, похожие на операции в функции `onEvent` в реализации самотестирования в данном примере.

Передача связанных методов в очередь

Технически, чтобы обеспечить еще более высокую гибкость этой схемы, в приложении PyMailGUI из главы 14 с помощью этого модуля в очередь будут помещаться *связанные методы* – вызываемые объекты, которые, как упоминалось, хранят ссылку на функцию метода и на экземпляр объекта, что обеспечивает возможность доступа к данным объекта и другим его методам. При таком подходе программный код управления потоками выполнения в клиентском сценарии выглядит примерно так, как показано в примере 10.21 – версии реализации самотестирования из предыдущего примера, использующей классы и методы.

Пример 10.21. PP4E\Gui\Tools\threadtools-test-classes.py

```
# тест очереди обработчиков, но для реализации операций используются
# связанные методы

import time
from threadtools import threadChecker, startThread
from tkinter.scrolledtext import ScrolledText

class MyGUI:
    def __init__(self, reps=3):
        self.reps = reps               # используется окно Tk по умолчанию
        self.text = ScrolledText()     # сохранить виджет в атрибуте
        self.text.pack()
        threadChecker(self.text)       # запустить цикл проверки потоков
        self.text.bind('<Button-1>',   # в 3.x функция list необходима для
            lambda event: list(map(self.onEvent, range(6))) ) # получения всех
                                       # результатов map, для range - нет

    def onEvent(self, i):              # метод, запускающий поток
        myname = 'thread-%s' % i
        startThread(
            action     = self.threadaction,
            args       = (i, ),
            context    = (myname,),
            onExit     = self.threadexit,
            onFail     = self.threadfail,
            onProgress = self.threadprogress)

    # основная операция, выполняемая потоком
    def threadaction(self, id, progress):  # то, что делает поток
        for i in range(self.reps):         # доступ к данным в объекте
            time.sleep(1)
            if progress: progress(i)       # обработчик progress: в очередь
        if id % 2 == 1: raise Exception    # ошибочный номер: неудача

    # обработчики: передаются главному потоку через очередь
    def threadexit(self, myname):
        self.text.insert('end', '%s\texit\n' % myname)
        self.text.see('end')
```

```
        def threadfail(self, exc_info, myname): # имеет доступ к данным объекта
            self.text.insert('end', '%s\tfail\t%s\n' % (myname, exc_info[0]))
            self.text.see('end')

        def threadprogress(self, count, myname):
            self.text.insert('end', '%s\tprog\t%s\n' % (myname, count))
            self.text.see('end')
            self.text.update() # допустимо: выполняется в главном потоке

if __name__ == '__main__': MyGUI().text.mainloop()
```

В этой версии в качестве обработчиков завершения и информирования о ходе выполнения задания, помещаемых в очередь, а также основной операции, выполняемой потоком, используются связанные методы. Как мы узнали в главе 5, благодаря тому что все потоки выполняются в пределах одного и того же процесса и в одной и той же области памяти, связанные методы ссылаются на оригинальные экземпляры объектов, а не на их копии. Это позволяет им напрямую обновлять графический интерфейс и другую информацию. Кроме того, связанные методы являются нормальными вызываемыми объектами, которые могут использоваться взамен обычных функций, поэтому нет никаких препятствий к использованию их в очередях и в потоках выполнения. Помимо всего прочего, широкие возможности совместного использования данных являются одним из основных преимуществ потоков выполнения перед процессами.

Приложение PyMailGUI в главе 14 демонстрирует более практичное применение этого модуля, где он служит базовым механизмом обслуживания событий завершения потоков и информирования о ходе выполнения задания. Там основные операции в потоках также выполняются с помощью связанных методов, что позволяет потокам и обработчикам, помещаемым ими в очередь, использовать одни и те же данные. Как мы увидим далее, действия, выполняемые обработчиками из очереди, автоматически становятся безопасными в многопоточном окружении, потому что они выполняются только в контексте главного потока. Однако другие изменения в совместно используемых объектах, производимые дочерними потоками, все еще может потребоваться синхронизировать отдельно, если они выполняются без применения очереди обработчиков и есть вероятность, что они будут перекрываться во времени. Непосредственное обновление кэша электронной почты, например, может заблокировать выполнение других операций до его завершения.

Другие способы добавления GUI к сценариям командной строки

Иногда потребность в графическом интерфейсе возникает совершенно неожиданно. Возможно, вы еще не умеете программировать графические интерфейсы или просто хотите оставаться в старых добрых време-

нах. Но, предположим, вы написали программу, взаимодействующую с пользователем посредством консоли, а потом решили, что взаимодействие через графический интерфейс сделает ее привлекательнее. Что делать в этом случае?

Вероятно, наиболее очевидный ответ состоит в том, чтобы преобразовать программу командной строки, – добавить в программный код инициализацию виджетов при запуске, вызвать функцию `mainloop`, чтобы запустить цикл обработки событий и отобразить главное окно, и переместить всю логику программы в обработчики, запускаемые в ответ на действия пользователя. Все операции, выполняемые оригинальной программой, превращаются в обработчики событий, а главный поток управления конструирует главное окно, вызывает цикл событий один раз и переходит в состояние ожидания.

Это традиционный способ организации программ с графическим интерфейсом, и он полностью соответствует ожиданиям пользователя – окна появляются по запросу, а не случайным, на первый взгляд, образом. Однако если вы не готовы стиснуть зубы и выполнить такое структурное преобразование, можно пойти другим путем. Например, в разделе, посвященном примеру `ShellGui`, выше в этой главе, мы видели, как можно добавить к сценариям архивирования файлов окна для ввода входных параметров (пример 10.5 и следующие за ним). Позднее мы также видели, как перенаправлять вывод таких сценариев в графические интерфейсы с помощью класса `GuiOutput` (пример 10.13). Этот подход можно использовать, когда сценарий командной строки, обертываемый в графический интерфейс, выполняет единственную операцию. Для организации более динамичного взаимодействия с пользователем может потребоваться использовать другие приемы.

Вполне возможно, например, запускать графический интерфейс из сценария командной строки, вызывая функцию `mainloop` из библиотеки tkinter всякий раз когда требуется отобразить окно. Возможно также использовать более фундаментальный подход и создать отдельную программу, реализующую графический интерфейс для вашего приложения. В заключение нашего обзора методик программирования графических интерфейсов мы по очереди рассмотрим все эти схемы.

Вывод окон графического интерфейса по требованию

Если вам необходимо добавить простой графический интерфейс для взаимодействия с пользователем к существующему сценарию командной строки (например, диалог выбора файла), это можно сделать, настроив виджеты и вызвав функцию `mainloop` из программы командной строки, когда это будет необходимо. По сути, этот прием добавляет поддержку графического интерфейса в программу, не имеющую постоянного главного окна. Проблема в том, что функция `mainloop` не возвращает управление, пока главное окно не будет закрыто пользователем (или не будет вызван метод `quit`). Поэтому вы не сможете получить данные,

введенные пользователем, из виджетов, уже уничтоженных к моменту возврата из функции `mainloop`. Чтобы обойти эту проблему, достаточно просто сохранить ввод пользователя в объекте Python: объект останется существовать после уничтожения графического интерфейса. Пример 10.22 демонстрирует один из способов реализации этой идеи на языке Python.

Пример 10.22. PP4E\Gui\Tools\mainloopdemo.py

```
"""
демонстрирует запуск двух отдельных циклов mainloop; каждый из них возвращает
управление после того как главное окно будет закрыто; ввод пользователя
сохраняется в объекте Python перед тем, как графический интерфейс будет
закрыт; обычно в программах с графическим интерфейсом настройка виджетов
и вызов mainloop выполняется всего один раз, а вся их логика распределена
по обработчикам событий; в этом демонстрационном примере вызовы функции
mainloop производятся для обеспечения модальных взаимодействий с пользователем
из программы командной строки; демонстрирует один из способов добавления
графического интерфейса к существующим сценариям командной строки без
реорганизации программного кода;
"""

from tkinter import *
from tkinter.filedialog import askopenfilename, asksaveasfilename

class Demo(Frame):
    def __init__(self,parent=None):
        Frame.__init__(self,parent)
        self.pack()
        Label(self, text ="Basic demos").pack()
        Button(self, text='open', command=self.openfile).pack(fill=BOTH)
        Button(self, text='save', command=self.savefile).pack(fill=BOTH)
        self.open_name = self.save_name = ""
    def openfile(self):                     # сохранить результаты пользователя
        self.open_name = askopenfilename() # указать параметры диалога здесь
    def savefile(self):
        self.save_name = asksaveasfilename(initialdir='C:\\Python31')

if __name__ == "__main__":
    # вывести окно
    print('popup1...')
    mydialog = Demo()          # присоединить фрейм к окну Tk() по умолчанию
    mydialog.mainloop()        # отобразить; вернуться после закрытия окна
    print(mydialog.open_name)  # имена сохраняются в объекте, когда окно уже
    print(mydialog.save_name)  # будет закрыто
    # Раздел программы без графического интерфейса, использующей mydialog

    # отобразить окно еще раз
    print('popup2...')
    mydialog = Demo()          # повторно создать виджеты
    mydialog.mainloop()        # повторно отобразить окно
```

```
print(mydialog.open_name)  # в объекте будут сохранены новые значения
print(mydialog.save_name)
# Раздел программы без графического интерфейса,
# где снова используется mydialog
print('ending...')
```

Эта программа дважды конструирует и отображает простое окно с двумя кнопками, как показано на рис. 10.13, нажатие которых вызывает появление диалогов выбора файла. Вывод программы, который производится при закрытии окна графического интерфейса, выглядит примерно, как показано ниже:

```
C:\...\PP4E\Gui\Tools> mainloopdemo.py
popup1...
C:/Users/mark/Stuff/Books/4E/PP4E/dev/Examples/PP4E/Gui/Tools/widgets.py
C:/Python31/python.exe
popup2...
C:/Users/mark/Stuff/Books/4E/PP4E/dev/Examples/PP4E/Gui/Tools/guimixin.py
C:/Python31/Lib/tkinter/__init__.py
ending...
```

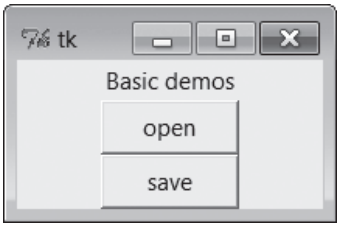


Рис. 10.13. *Окно, которое выводится программой командной строки*

Обратите внимание, что в этой программе функция `mainloop` вызывается дважды, для организации двух модальных взаимодействий с пользователем из сценария командной строки, не имеющего графического интерфейса. Нет ничего криминального в том, что функция `mainloop` вызывается более одного раза, но при этом сценарию приходится заново создавать виджеты перед очередным вызовом, потому что они уничтожаются после предыдущего вызова `mainloop` (виджеты уничтожаются внутри библиотеки Tk, даже если соответствующие им объекты Python продолжают существование). Напомню, что такая реализация графического интерфейса не соответствует ожиданиям пользователей, в сравнении с традиционными графическими интерфейсами, – создается впечатление, что окна появляются из ниоткуда, – но это самый быстрый способ добавить графический интерфейс без глубокой реорганизации программного кода.

Обратите внимание, что данный пример отличается от случая использования вложенных (рекурсивных) вызовов функции `mainloop` для ре-

ализации модальных диалогов, с которым мы столкнулись в главе 8. При таком подходе вложенные вызовы `mainloop` возвращают управление, когда вызывается метод `quit` диалога, но при этом продолжается выполнение объемлющего вызова `mainloop`, и мы остаемся в сфере программирования, управляемого событиями. Сценарий в примере 10.22, напротив, производит два независимых вызова функции `mainloop`, дважды вступая и выходя из модели, управляемой событиями.

Наконец, обратите внимание, что такая схема подходит только для ситуаций, когда не требуется выполнять какие-либо операции, не связанные с графическим интерфейсом, пока окно остается открытым, потому что на период выполнения `mainloop` основной поток управления сценария остается неактивным и блокируется. Вы не сможете, например, применить этот подход для добавления графического интерфейса к утилитам, подобным тем, что используются в модуле `guiStreams`, представленном выше в этой главе, предназначенном для передачи функций взаимодействия с пользователем из сценариев командной строки в графический интерфейс. Классы `GuiInput` и `GuiOutput` в том примере предполагают, что где-то уже был произведен вызов `mainloop` (в конце концов, они опираются на использование графического интерфейса). Но как только будет вызвана функция `mainloop`, чтобы вывести окна, вы не сможете вернуть управление обычному программному коду сценария командной строки, чтобы взаимодействовать с пользователем или с графическим интерфейсом, пока этот графический интерфейс не будет закрыт и функция `mainloop` не вернет управление. Таким образом, эти классы могут использоваться только в контексте программ, полностью опирающихся на графический интерфейс.

Но на самом деле это неестественный способ использования библиотеки tkinter. Сценарий в примере 10.22 действует только потому, что графический интерфейс может взаимодействовать с пользователем совершенно независимо, – сценарий может позволить себе отдать управление функции `mainloop` из библиотеки tkinter и ждать результатов. Эта схема непригодна, когда требуется выполнять программный код, не имеющий отношения к графическому интерфейсу, в то время, когда окно остается открытым. Из-за этих ограничений в большинстве графических интерфейсов вам придется использовать модель главное-окно-плюс-обработчики-событий – обработчики вызываются в ответ на действия пользователя, пока окно графического интерфейса остается открытым. При таком подходе ваш программный код может действовать, пока окно остается открытым. Например, смотрите представленный ранее в этой главе способ запуска сценариев командной строки архивирования и разархивирования из графического интерфейса, с выводом результатов в графическом интерфейсе, – технически эти сценарии запускаются из обработчиков событий графического интерфейса, а их вывод перенаправляется в виджет.

Реализация графического интерфейса в виде отдельной программы: сокеты (вторая встреча)

Как отмечалось ранее, возможно также реализовать графический интерфейс приложения в виде отдельной программы. Это наиболее тернистый путь, но в некоторых ситуациях он может упростить интеграцию слабо связанных компонентов. Этот способ может, например, помочь решить проблемы, свойственные примеру `guiStreams` из предыдущего раздела, при условии, что входные и выходные данные будут передаваться графическому интерфейсу через механизмы взаимодействий между процессами (Inter-Process Communication, IPC), а для обнаружения выходных данных будет использован метод `after` виджетов (или подобный ему). В этом случае работа сценария командной строки не блокируется вызовом функции `mainloop`.

Графический интерфейс может запускаться сценарием командной строки как отдельная программа, и обмениваться результатами взаимодействий с пользователем с основным сценарием посредством каналов, сокетов, файлов или других механизмов IPC, представленным в главе 5. Преимущество такого подхода состоит в том, что он обеспечивает разделение представления и реализации – в сценарий, который иначе может использоваться как обычная утилита командной строки, достаточно будет добавить всего лишь запуск графического интерфейса и организовать прием ввода пользователя. Кроме того, работа сценария командной строки не будет блокироваться на время работы функции `mainloop` (функция `mainloop` будет выполняться только в процессе, реализующем графический интерфейс), а сам графический интерфейс может сохраняться на экране и после того, как пользователь введет все необходимые данные, что позволит уменьшить количество всплывающих окон.

Другой вариант – когда графический интерфейс запускает сценарий командной строки и организует прием данных от него с применением механизмов IPC, подключаемых к потоку стандартного вывода сценария. В еще более сложных решениях графический интерфейс и сценарий командной строки могут организовать двусторонний обмен данными.

Примеры 10.23, 10.24 и 10.25 демонстрируют простые варианты реализации этих подходов: вывод сценария командной строки отправляется в графический интерфейс. В них представлены реализации сценариев командной строки и графического интерфейса, которые взаимодействуют друг с другом через *сокеты* – механизм сетевых взаимодействий, который коротко рассматривался в главе 5 и подробно будет исследоваться в следующей части книги. При изучении этих файлов особое внимание обратите на то, как организована связь между программами: когда сценарий командной строки выводит что-то в стандартный поток вывода, текст отправляется графическому интерфейсу через сетевое соединение. Кроме того, что он импортирует модуль и вызывает функцию перенаправления вывода в сокет, сценарий командной строки ничего не знает ни о графических интерфейсах, ни о сокетах, а графический

интерфейс ничего не знает о сценарии, вывод которого он отображает. Так как эта модель не требует полностью переписывать существующие сценарии, чтобы добавить к ним поддержку графического интерфейса, она является идеальной для сценариев, которые живут и действуют в мире командных оболочек и командной строки.

С точки зрения реализации, нам сначала необходимо создать механизм IPC, который свяжет сценарий с графическим интерфейсом. Пример 10.23 содержит реализацию подключения к сокетам на стороне клиента, используемую сценарием командной строки. В данный момент представлена только частичная реализация модуля (обратите внимание на оператор многоточия `...` в последних нескольких функциях – своего рода фразу «Подлежит реализации» на языке Python; этот оператор является эквивалентом инструкции `pass` в данном контексте). Так как подробно сокеты будут рассматриваться только в главе 12, мы отложим реализацию других режимов перенаправления до этого момента; там также будет представлена остальная часть реализации этого модуля. Версия модуля, представленная здесь, реализует перенаправление в сокет только потока стандартного вывода и отлично подходит для графического интерфейса, которому требуется перехватить вывод сценария командной строки.

Пример 10.23. PP4E\Gui\Tools\socket_stream_redirect0.py

```
"""
[частичная реализация] Инструменты подключения потоков ввода-вывода сценариев
командной строки к сокетам, которые могут использоваться графическими
интерфейсами (и другими сценариями) для взаимодействий с этими сценариями; более
полное обсуждение и реализацию вы найдете в главе 12 и в каталоге PP4E\Sockets\
Internet
"""

import sys
from socket import *
port = 50008
host = 'localhost'

def redirectOut(port=port, host=host):
    """
    подключает стандартный поток вывода вызывающего сценария к сокету, для
    передачи данных графическому интерфейсу, прослушивающему сетевое
    соединение;
    вызывающий сценарий должен запускаться после того как будет запущен
    сценарий или графический интерфейс, прослушивающий сетевое соединение,
    иначе connect потерпит неудачу до того, как будет выполнена функция accept
    """
    sock = socket(AF_INET, SOCK_STREAM)
    sock.connect((host, port))# вызывающий сценарий играет роль клиента
    file = sock.makefile('w') # интерфейс файлов: текстовый режим, буферизация
    sys.stdout = file         # заставить функцию print выводить текст
                              # с помощью sock.send
```

```
def redirectIn(port=port, host=host): ...           # см. главу 12
def redirectBothAsClient(port=port, host=host): ... # см. главу 12
def redirectBothAsServer(port=port, host=host): ... # см. главу 12
```

Далее, пример 10.24 использует пример 10.23 для перенаправления потока стандартного вывода в сетевое соединение, которое может прослушиваться серверной программой, реализующей графический интерфейс. Для этого требуется добавить в начало сценария всего две строки программного кода, которые выполняются в зависимости от наличия аргумента командной строки (если сценарий запускается без аргументов, он выполняется в обычном режиме):

Пример 10.24. PP4E\Gui\Tools\socket-nongui.py

```
# сценарий командной строки: подключает поток вывода к сокету
# и действует как обычно

import time, sys
if len(sys.argv) > 1:        # подключаться к GUI только при явном требовании
    from socket_stream_redirect0 import * # подключить sys.stdout к сокету
    redirectOut()                         # GUI должен запускаться первым

# программный код, не связанный с графическим интерфейсом
while True:                           # выводит данные в stdout:
    print(time.asctime())             # передать процессу GUI через сокет
    sys.stdout.flush()                # требуется для передачи: буферизация!
    time.sleep(2.0)                   # небуферизованный режим отсутствует
                                      # ключ -u не решает проблему
```

И наконец, в примере 10.25 приводится реализация графического интерфейса, участвующего в обмене данными. Этот сценарий создает графический интерфейс для отображения текста, выводимого программой командной строки, но он ничего не знает о логике работы другой программы. Для отображения получаемого текста графический интерфейс использует объект перенаправления потока вывода в виджет, с которым мы встречались выше в этой главе (пример 10.12), – поскольку данная программа вызывает функцию `mainloop`, этот объект «просто работает».

Кроме того, для проверки наличия входных данных в сокете здесь используется цикл обработки событий от таймера, вместо того чтобы ждать завершения программы командной строки. Поскольку сокет настраивается на работу в *неблокирующем* режиме, операция ввода не ждет, пока появятся данные, и не блокирует графический интерфейс.

Пример 10.25. PP4E\Gui\Tools\socket-gui.py

```
# сервер GUI: читает и отображает текст, полученный
# от сценария командной строки

import sys, os
from socket import *           # включая socket.error
from tkinter import Tk
from PP4E.launchmodes import PortableLauncher
```

```
from PP4E.Gui.Tools.guiStreams import GuiOutput

myport = 50008
sockobj = socket(AF_INET, SOCK_STREAM) # GUI - сервер, сценарий - клиент
sockobj.bind((‘’, myport))             # сервер настраивается перед
sockobj.listen(5)                      # запуском клиента

print(‘starting’)
PortableLauncher(‘nongui’, ‘socket-nongui.py -gui’)() # запустить сценарий

print(‘accepting’)
conn, addr = sockobj.accept()          # ждать подключения клиента
conn.setblocking(False)                # неблокирующий сокет (False=0)
print(‘accepted’)

def checkdata():
    try:
        message = conn.recv(1024)      # попытка ввода не блокируется
        #output.write(message + ‘\n’)  # можно также сделать sys.stdout=output
        print(message, file=output)    # если текст получен - вывести в окне
    except error:                      # возбудит socket.error, если нет данных
        print(‘no data’)               # вывести в sys.stdout
    root.after(1000, checkdata)        # проверять раз в секунду

root = Tk()
output = GuiOutput(root)               # текст из сокета отображается здесь
checkdata()
root.mainloop()
```

Запустите сценарий из примера 10.25, чтобы протестировать весь комплекс. Когда запустятся оба процесса, графический интерфейс и сценарий командной строки, графический интерфейс примерно каждые две секунды будет получать через сокет новые сообщения и отображать их в окне, как показано на рис. 10.14. Цикл обработки событий от таймера в графическом интерфейсе проверяет поступление новых данных примерно раз в секунду, но сценарий командной строки отправляет сообщения только раз в две секунды, из-за задержки, организованной с помощью функции `time.sleep`. Ниже приводится пример вывода в окно консоли – сообщения «no data» в консоли и новые строки в графическом интерфейсе появляются каждую секунду:

```
C:\...\PP4E\Gui\Tools> socket-gui.py
starting
nongui
accepting
accepted
no data
no data
no data
no data
```

...часть строк опущена...

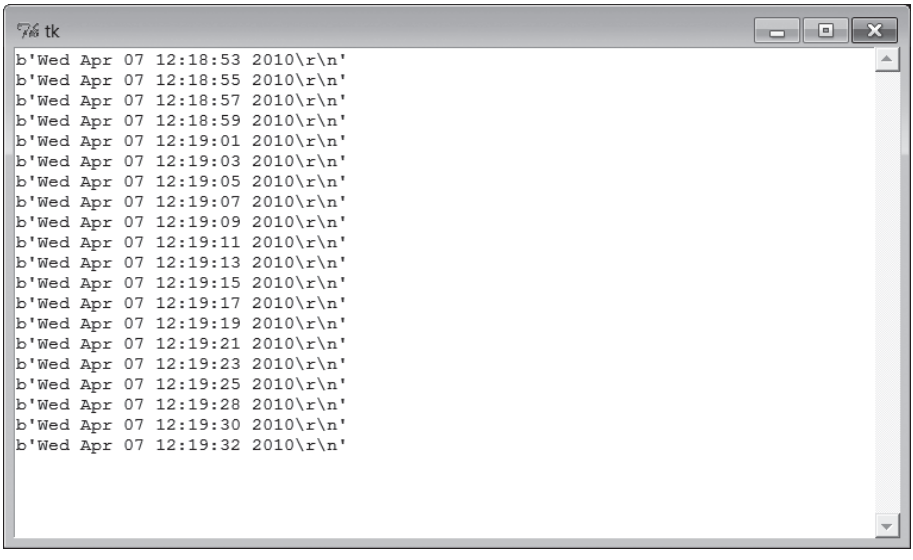


Рис. 10.14. *Сообщения от сценария командной строки, выводимые графическим интерфейсом (сокеты)*

Обратите внимание на рис. 10.14, что мы отображаем строки типа `bytes`, – несмотря на то, что сценарий командной строки выводит текст, сценарий графического интерфейса получает строки байтов, потому что читает их, используя низкоуровневый интерфейс сокетов, а сокеты в Python 3.X обрабатывают данные в виде строк двоичных байтов.

Запустите этот сценарий у себя на компьютере, чтобы посмотреть, как он действует. В общих чертах, сценарий графического интерфейса запускает сценарий командной строки и отображает окно, в которое выводит текст, печатаемый сценарием командной строки (дата и время). Сценарий командной строки может по-прежнему выполнять линейный процедурный программный код и воспроизводить данные, потому что только процесс графического интерфейса выполняет цикл событий `mainloop`.

Кроме того, в отличие от ранее исследованных нами приемов перенаправления, когда мы просто подключали потоки ввода-вывода сценария к объектам графического интерфейса, данный подход деления на два процесса предотвращает блокирование графического интерфейса в ожидании, пока сценарий выведет какие-либо данные. Процесс графического интерфейса остается полностью независимым и активным и просто извлекает новые результаты по мере их поступления (подробнее об этом рассказывается в следующем разделе). Данная модель по духу напоминает предыдущие примеры с потоками выполнения и очередями, только здесь главными действующими лицами являются отдельные программы, связанные с помощью сокета, а не вызовы функций в контексте единого процесса.

Мы не будем подробно рассматривать сокеты в этой главе, чтобы объяснить их применение в этом программном коде, тем не менее следует подчеркнуть несколько наиболее важных моментов:

- Вероятно, этот пример следовало бы дополнить возможностью определения признака конца файла, отправляемого дочерним сценарием, и завершать цикл обработки событий от таймера.
- Сценарий командной строки мог бы сам запускать графический интерфейс, но в мире сокетов серверный конец (графический интерфейс) должен быть настроен на прием входящих соединений *раньше*, чем клиент (сценарий командной строки) попытается соединиться с ним. Так или иначе, графический интерфейс должен быть запущен еще до того, как сценарий командной строки попытается установить соединение, иначе соединение не будет установлено и сценарий потерпит неудачу.
- Из-за поддержки буферизации в текстовом режиме, свойственной объектам `socket.makefile`, используемым здесь для перенаправления потока вывода, клиентская программа обязательно должна выталкивать выходной буфер с помощью `sys.stdout.flush`, чтобы отправить данные графическому интерфейсу, – без вызова этого метода графический интерфейс ничего не будет получать и отображать. Как будет показано в главе 12, этот прием не обязательно применять при использовании каналов, но обязательно – при работе с обертками сокетов, как в данном примере. В Python 3.X эти обертки не поддерживают небуферизованные режимы и не имеют ключа командной строки, такого как `-u`, для данного контекста (дополнительные сведения о ключе `-u` и о каналах приводятся в следующем разделе).

Дополнительную информацию к этому примеру и по данной теме вы найдете в главе 12. Модель клиент-сервер на основе сокетов неплохо подходит для соединения графического интерфейса со сценариями командной строки, но существуют и другие альтернативы, которые мы рассмотрим в следующем разделе, прежде чем двинуться дальше.

Реализация графического интерфейса в виде отдельной программы: каналы

Объединение двух программ в предыдущем разделе напоминает программу с графическим интерфейсом, которая читает вывод команды, запущенной с помощью `os.popen` (или с помощью интерфейса `subprocess.Popen`, который опирается на эту функцию). Как будет показано далее, сокеты также поддерживают возможность обмена данными с независимыми серверами и могут использоваться для соединения программ, выполняющихся на разных компьютерах в сети, однако эту идею мы будем рассматривать в главе 12.

Пожалуй, еще более тонким и важным для нашего исследования графических интерфейсов является тот факт, что без цикла обработки со-

бытий от таймера на основе метода after и неблокирующей операции чтения данных, подобной той, что использовалась в предыдущем разделе, графический интерфейс может блокироваться в ожидании поступления данных от программы командной строки и оказаться неспособным обрабатывать более одного потока данных.

Предлагаю взглянуть на функцию redirectedGuiShellCmd в примере 10.12, перенаправляющую вывод команды оболочки, запускаемой с помощью os.popen, в окно графического интерфейса. Мы могли бы использовать простейший программный код, как в примере 10.26, чтобы перехватить вывод порожденной программы на языке Python и отобразить его в окне отдельной программы с графическим интерфейсом. Решение получилось таким компактным благодаря тому, что оно опирается на цикл чтения/записи и на класс GuiOutput из примера 10.12 для управления графическим интерфейсом и чтения данных из канала. Это решение, по сути, повторяет один из вариантов реализации самотестирования в том примере, но здесь мы читаем вывод программы на языке Python.

Пример 10.26. PP4E\Gui\Tools\pipe-gui1.py

```
# графический интерфейс: перенаправляет стандартный вывод порождаемой
# программы в окно GUI

from PP4E.Gui.Tools.guiStreams import redirectedGuiShellCmd # исп-ет GuiOutput
redirectedGuiShellCmd('python -u pipe-nongui.py')           # -u: без
                                                            # буферизации
```

Обратите внимание на ключ -u командной строки интерпретатора Python, используемый здесь: он принудительно отключает буферизацию потока стандартного вывода запускаемой программы, поэтому мы получаем печатаемый текст немедленно и нам не приходится ждать завершения дочерней программы.

Мы говорили об этой возможности в главе 5, когда обсуждали каналы и состояния взаимоблокировки. Напомню, что функция print выводит текст в sys.stdout, который обычно предусматривает буферизацию при подключении к каналу таким способом. Если бы мы здесь не использовали ключ -u и порожденная программа не вызывала бы метод sys.stdout.flush, мы ничего не увидели бы в графическом интерфейсе, пока дочерняя программа не завершилась бы или пока не переполнился буфер. Если дочерняя программа выполняет бесконечный цикл, нам может потребоваться ждать очень долго, пока вывод появится в канале и, соответственно, в графическом интерфейсе.

Такой подход значительно упрощает реализацию сценария командной строки, как показано в примере 10.27: он просто выводит текст в стандартный поток вывода и ему не требуется выполнять подключение к сокету. Сравните его с эквивалентной реализацией на основе сокетов, представленной в примере 10.24, – цикл тот же самый, но здесь не требуется предварительно выполнять подключение к сокету (родительская программа читает обычный поток вывода) и нет необходимости

вручную выталкивать выходной буфер (ключ -u, указанный при запуске дочерней программы, отключает буферизацию).

Пример 10.27. PP4E\Gui\Tools\pipe-nongui.py

```
# сценарий командной строки: действует как обычно, не требует выполнения
# дополнительных операций
import time
while True:                # реализация сценария командной строки
    print(time.asctime()) # отправить процессу GUI
    time.sleep(2.0)       # выталкивать буфер здесь не требуется
```

Запустите сценарий графического интерфейса из примера 10.26: он автоматически запустит сценарий командной строки, подключится к его потоку стандартного вывода и отобразит окно, как показано на рис. 10.15. Своим внешним видом оно напоминает окно сценария, реализованного на основе сокетов, изображенное на рис. 10.14, но в данном случае будут выводиться строки str, которые мы получаем при чтении каналов, а не строки байтов, как при чтении из сокетов.

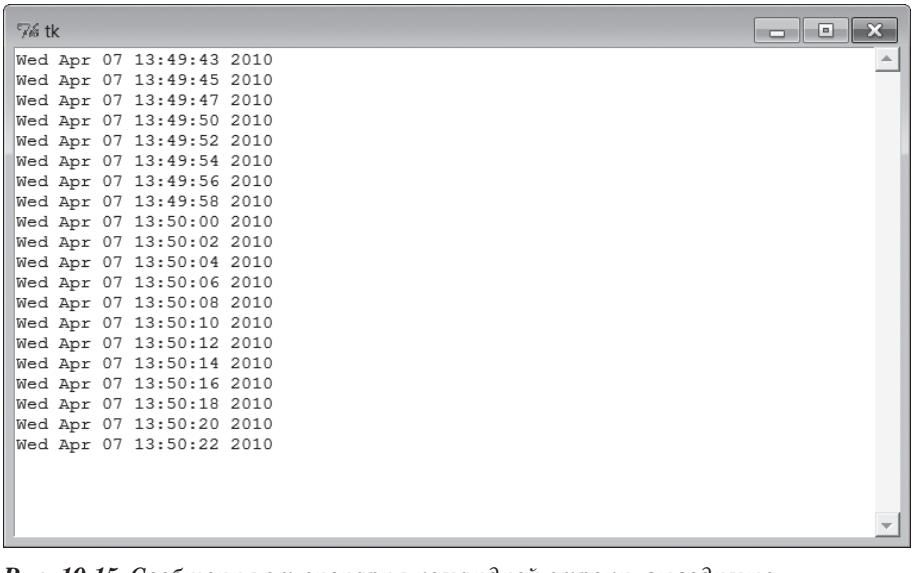


Рис. 10.15. *Сообщения от сценария командной строки, выводимые графическим интерфейсом (каналы)*

Сценарии действуют, но реализация графического интерфейса выглядит несколько странно – в ней отсутствует явный вызов функции mainloop, и мы получаем дополнительное пустое окно верхнего уровня по умолчанию. Фактически этот графический интерфейс действует лишь благодаря вызову метода update внутри функции перенаправления, который на мгновение передает управление циклу событий Tk, чтобы обработать ожидающие события. Более удачное решение представлено

в примере 10.28. Этот сценарий создает графический интерфейс и запускает цикл событий вручную до того, как будет запущена команда оболочки, – при ее запуске воспроизводится то же самое окно (рис. 10.15).

Пример 10.28. PP4E\Gui\Tools\pipe-gui2.py

```
# графический интерфейс: действует так же, как pipes-gui1, но явно создает
# главное окно и запускает цикл событий

from tkinter import *
from PP4E.Gui.Tools.guiStreams import redirectedGuiShellCmd

def launch():
    redirectedGuiShellCmd('python -u pipe-nongui.py')

window = Tk()
Button(window, text='GO!', command=launch).pack()
window.mainloop()
```

Ключ `-u`, отключающий буферизацию, здесь также имеет большое значение – без него мы не увидели бы текст в окне вывода. Графический интерфейс оказался бы заблокирован в первой операции чтения из канала, потому что текст, выводимый дочерним сценарием, оставался бы в буфере потока стандартного вывода.

С другой стороны, наличие ключа `-u`, запрещающего буферизацию, не предотвращает блокирование графического интерфейса из предыдущего раздела, использующего сокеты, потому что в том примере после запуска дочерней программы поток вывода переключается на другой объект. Дополнительные сведения об этом приводятся в главе 12. Запомните также, что аргумент функции `os.popen` (и `subprocess.Popen`), определяющий параметры буферизации, управляет буферизацией только на стороне вызывающего процесса, но не в порожденной программе, тогда как ключ `-u` передается при запуске последней.

Призрак блокирования операций чтения

Однако при любом подходе графические интерфейсы в примерах 10.26 и 10.28 оказываются заблокированными на две секунды каждый раз, когда пытаются прочитать данные из канала с помощью `os.popen`. На практике интерфейсы становятся очень неповоротливыми – команды переместить окно, изменить его размер, перерисовать, поднять над другими окнами и так далее ожидают до двух секунд, пока сценарий командной строки не отправит данные графическому интерфейсу и тем самым не обеспечит возврат из функции чтения канала. Еще хуже то, что если щелкнуть на кнопке `GO!` дважды во второй версии графического интерфейса, только одно окно будет обновляться каждые две секунды, потому что графический интерфейс «застрянет» в обработчике события нажатия кнопки – он не сможет выйти из цикла чтения, пока дочерний сценарий командной строки не завершится. Завершение ра-

боты программы также выполняется не очень изящно (в окне консоли появится множество сообщений об ошибках).

Из-за этих ограничений – чтобы избежать заблокированных состояний – независимо запускаемый графический интерфейс не должен читать данные непосредственно, если могут возникать задержки в отображении. Например, в сценарии из предыдущего раздела (пример 10.25), использующем сокеты, цикл обработки событий от таймера `after` позволяет графическому интерфейсу проверять наличие данных вместо того, чтобы *ждать* их, и отображать по мере появления. Поскольку графический интерфейс не ждет, пока данные появятся, он остается активным между операциями вывода.

Конечно, истинная причина этих проблем заключается в том, что цикл чтения/записи в используемой здесь функции из модуля `guiStreams` слишком упрощен – ошибочное размещение операции чтения в графическом интерфейсе провоцирует блокировку. Существуют различные решения, позволяющие избежать этого.

Обновление графических интерфейсов внутри потоков выполнения... и другие «решения»

Чтобы исправить эту проблему, можно было бы попробовать вызывать функцию перенаправления в дочернем потоке выполнения, например, изменив функцию `launch` в примере 10.28, как показано ниже (этот фрагмент взят из сценария *pipe-gui2-thread.py*, входящего в состав пакета с примерами для книги):

```
def launch():
    import _thread
    _thread.start_new_thread(redirectedGuiShellCmd,
                             ('python -u pipe-nongui.py',))
```

Но тогда графический интерфейс будет обновляться из дочернего потока выполнения, что, как мы уже знаем, заканчивается плохо. Параллельные попытки обновления графического интерфейса могут нанести ему вред.

Фактически, после внесения предложенных изменений на моем ноутбуке с Windows 7, графический интерфейс зависает сразу же после первого щелчка на кнопке `GO!`, становясь совершенно неотзывчивым, и его приходится закрывать принудительно. Это происходит до (или, может быть, во время) создания нового окна с текстовым виджетом. Когда этот пример запускался в Windows XP, во время работы над предыдущим изданием книги, он также иногда подвисал при первом щелчке на кнопке `GO!`, а несколько щелчков на кнопке гарантированно подвешивали его и в этой системе – процесс приходилось останавливать принудительно. Прямое обновление графического интерфейса из дочерних потоков выполнения не является приемлемым решением.

Как вариант, можно было бы попробовать использовать функцию Python `select.select` (описывается в главе 12) для реализации проверки наличия данных в канале – к сожалению, в настоящее время функция `select` в Windows работает только с сокетами (в Unix она также работает с каналами и с дескрипторами файлов).

В некоторых контекстах графический интерфейс, запускаемый отдельно, мог бы использовать сигналы для информирования программы командной строки о наступлении момента обмена данными, и наоборот (с помощью модуля `signal` и функции `os.kill`, представленных в главе 5). Недостаток такого решения состоит в том, что он требует добавлять обработку сигналов в реализацию программы командной строки.

Альтернативой сокетам в примерах с 10.23 по 10.25 могли бы в определенных ситуациях служить именованные каналы (файлы fifo, представленные в главе 5), но сокеты работают в стандартной версии Python для Windows, а именованные каналы – нет (функция `os.mkfifo` недоступна в версии Python 3.1 для Windows, хотя она имеется в версии Cygwin Python). Но даже там, где они работают, нам все еще необходимо использовать цикл обработки событий от таймера на основе метода `after`, чтобы избежать блокирования графического интерфейса.

Также мы могли бы использовать функцию `createfilehandler` из библиотеки tkinter, чтобы зарегистрировать обработчик, который будет вызываться при появлении данных в канале:

```
def callback(file, mask):
    ...чтение данных из файла...

import _tkinter, tkinter
_tkinter.createfilehandler(file, tkinter.READABLE, callback)
```

Операция регистрации обработчика доступна в виде функции в модуле `tkinter` и в виде метода экземпляра класса `Tk`. К сожалению, как уже отмечалось в конце главы 9, эта функция недоступна в Windows и может служить альтернативой только в Unix.

Предотвращение блокирования операций чтения с помощью потоков выполнения

Намного более универсальное решение проблемы блокирования операций чтения заключается в том, чтобы графический интерфейс порождал дочерний поток, который будет читать данные из сокета или канала и помещать их в очередь. Фактически прием, основанный на потоках выполнения, с которым мы встречались выше в этой главе, можно было бы напрямую использовать для решения данной проблемы. При таком подходе, пока поток выполнения ждет поступления данных, графический интерфейс не блокируется, а поток выполнения не пытается обновлять графический интерфейс. Кроме того, одновременно могут выполняться несколько потоков и производить продолжительные операции.

Пример 10.29 демонстрирует реализацию этого решения. Основная хитрость состоит в том, чтобы отделить операции ввода и вывода в оригинальной функции `redirectedGuiShellCmd` из модуля `guiStreams`, представленного в примере 10.12. В той версии операция ввода запускается в параллельном потоке выполнения и не блокирует графический интерфейс. Главный поток графического интерфейса использует цикл обработки событий от таймера `after` как обычно – чтобы проверять наличие данных в общей очереди, добавляемых потоком чтения. Так как главный поток сам не занимается чтением вывода дочерней программы, он не блокируется в ожидании поступления новых данных.

Пример 10.29. PP4E\Gui\Tools\pipe_gui3.py

```
"""
читает данные из канала в отдельном потоке выполнения и помещает их в очередь,
которая проверяется в цикле обработки событий от таймера; позволяет сценарию
отображать вывод программы, не вызывая блокирование графического интерфейса
между операциями вывода; со стороны дочерних программ не требуется выполнять
подключение или выталкивать буферы, но данное решение сложнее, чем подход на
основе сокетов
"""

import _thread as thread, queue, os
from tkinter import Tk
from PP4E.Gui.Tools.guiStreams import GuiOutput
stdoutQueue = queue.Queue()        # бесконечной длины

def producer(input):
    while True:
        line = input.readline()    # блокирование не страшно: дочерний поток
        stdoutQueue.put(line)      # пустая строка - конец файла
        if not line: break

def consumer(output, root, term='<end>'):
    try:
        line = stdoutQueue.get(block=False) # главный поток: проверять очередь
    except queue.Empty:                     # 4 раза в сек, это нормально,
        pass                                # если очередь пуста
    else:
        if not line:               # остановить цикл по достижении конца файла
            output.write(term)     # иначе отобразить следующую строку
            return
        output.write(line)
    root.after(250, lambda: consumer(output, root, term))

def redirectedGuiShellCmd(command, root):
    input = os.popen(command, 'r') # запустить программу командной строки
    output = GuiOutput(root)
    thread.start_new_thread(producer, (input,))  # запустить поток чтения
    consumer(output, root)
```

```
if __name__ == '__main__':
    win = Tk()
    redirectedGuiShellCmd('python -u pipe-nongui.py', win)
    win.mainloop()
```

Здесь мы используем очередь, чтобы избежать необходимости обновления графического интерфейса в дочерних потоках. Обратите внимание, что в предыдущем разделе, в примере с сокетами, очереди и потоки выполнения не требовались лишь потому, что у нас была возможность проверить сокет на наличие данных без блокирования – цикла обработки событий от таймера `after` было вполне достаточно. Однако при организации обмена данными через канал потоки выполнения являются самым простым способом избежать блокирования графического интерфейса.

Если запустить этот сценарий, программный код самотестирования создаст окно с виджетом `ScrolledText`, в котором будут отображаться текущие дата и время, отправляемые сценарием *pipes-nongui.py* из примера 10.27. Фактически это окно идентично тем, что создают предыдущие версии (рис. 10.15). Каждые две секунды в окне будет появляться новая строка, потому что именно с такой частотой сценарий `pipes-nongui` выводит сообщения в `stdout`.

Обратите внимание, что поток-производитель загружает данные по одной строке с помощью метода `readline()`. Мы не можем использовать функции чтения, которые пытаются загрузить все данные из потока ввода целиком (такие как `read()`, `readlines()`), потому что они не возвращают управление, пока программа не завершится и не отправит признак конца файла. Для чтения фрагмента данных можно было бы использовать метод `read(N)`, но в этом случае мы исходим из предположения, что в поток вывода передаются текстовые данные. Обратите также внимание, что здесь снова используется ключ `-u`, запрещающий буферизацию потоков ввода-вывода, чтобы обеспечить получение данных по мере их вывода. Без этого выводимые данные вообще не попали бы в графический интерфейс, потому что сохранялись бы в выходном буфере дочерней программы (попробуйте сами).

Сокеты и каналы: сходства и различия

Давайте посмотрим, что у нас получилось. Этот сценарий по своему духу напоминает сценарий в примере 10.28. Тем не менее, благодаря реструктуризации программного кода, сценарий в примере 10.29 имеет значительное преимущество: так как на этот раз операция чтения данных выполняется в дочернем *потоке*, графический интерфейс остается отзывчивым. Операции перемещения окна, изменения его размеров и так далее, выполняются немедленно, потому что графический интерфейс не блокируется в ожидании вывода очередной порции данных программой командной строки. Комбинация канала, потока выполнения и очереди в этом примере творит чудеса – графическому интерфей-

су не приходится ждать дочернюю программу, а дочернему потоку не требуется обновлять графический интерфейс.

Несмотря на сложность реализации и необходимость использовать многопоточную модель выполнения, отсутствие блокировок в примере 10.29 делают его функцию `redirectedGuiShellCmd` намного более полезной, чем в оригинальной версии. Тем не менее, в сравнении с реализацией на основе *сокетов* из предыдущего раздела, данное решение выглядит как смесь разных приемов:

- Поскольку эта реализация графического интерфейса читает данные из стандартного потока вывода дочерней программы, отпадает необходимость вносить в нее какие-либо изменения. В отличие от примера из предыдущего раздела, основанного на применении сокетов, программе командной строки не требуется знать о существовании графического интерфейса, отображающего ее результаты, – ей не требуется выполнять подключение к сокету и выталкивать свои выходные буферы, как в предыдущем решении с сокетами.
- Несмотря на отсутствие необходимости вносить изменения в программу, вывод которой отображается, сложность реализации графического интерфейса начинает приближаться к сложности реализации варианта на основе сокетов, особенно если отбросить шаблонный программный код, необходимый в любой программе, использующей сокеты.
- Данное решение не поддерживает возможность выполнения графического интерфейса и программы командной строки независимо друг от друга или на разных компьютерах. Как мы увидим в главе 12, сокеты позволяют передавать данные между программами, работающими на одном и том же компьютере, или по сети.
- Сокеты могут применяться не только для отображения стандартного потока вывода программы. Если от графического интерфейса требуется нечто большее, чем отображение вывода другой программы, сокеты могут обеспечить более универсальное решение. Кроме того, как мы увидим далее, сокеты по своей природе являются двунаправленными потоками данных, поэтому они позволяют передавать данные между программами в обоих направлениях более произвольными способами.

Другие примеры использования многопоточных графических интерфейсов и каналов

Несмотря на некоторые незначительные недостатки, реализация графических интерфейсов на основе потоков/очередей/каналов имеет весьма широкую область применения. Для иллюстрации приведем еще один короткий пример использования. Ниже демонстрируется запуск простого сценария в окне консоли, который каждые две секунды выводит все более и более длинную строку:

```
C:\...\PP4E\Gui\Tools> type spams.py
import time
for i in range(1, 10, 2):
    time.sleep(2)              # выводит текст в стандартный поток вывода
    print('spam' * i)          # GUI ничего не знает об этом, ведь так?

C:\...\PP4E\Gui\Tools> python spams.py
spam
spamspamspam
spamspamspamspamspam
spamspamspamspamspamspamspam
spamspamspamspamspamspamspamspamspam
```

Попробуем завернуть этот сценарий в графический интерфейс, введя программный код в интерактивной оболочке, для разнообразия. Следующий фрагмент импортирует новую версию функции перенаправления потока вывода в графический интерфейс как библиотечный компонент и с ее помощью создает окно, в котором отображаются пять строк, выводимые сценарием каждые две секунды – так же, как в окне консоли, – за которыми следует строка <end>, отражающая момент завершения дочерней программы. Получившееся окно изображено на рис. 10.16:

```
C:\...\PP4E\Gui\Tools> python
>>> from tkinter import Tk
>>> from pipe_gui3 import redirectedGuiShellCmd
>>> root = Tk()
>>> redirectedGuiShellCmd('python -u spams.py', root)
```

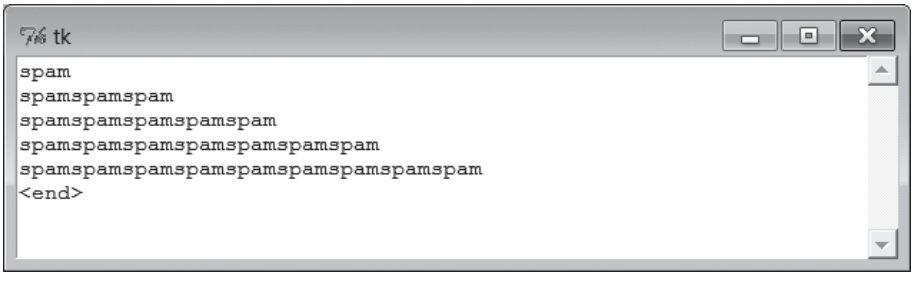


Рис. 10.16. *Графический интерфейс, отображающий полученный по каналу вывод другой программы*

Когда дочерняя программа завершится, поток-производитель в примере 10.29 определит признак конца файла и поместит в очередь заключительную пустую строку. В ответ на это цикл обработки событий от таймера выведет строку <end> в окно. В данном случае программа завершается обычным образом без вывода каких-либо сообщений, но в других ситуациях нам может потребоваться добавить логику завершения, чтобы подавить вывод сообщений об ошибках. Обратите внимание, что здесь, как и прежде, дочерняя программа имитирует выполнение про-

должительных операций с помощью функции `sleep`, а кроме того, нам необходимо использовать ключ `-u`, чтобы запретить буферизацию стандартного потока вывода, – без этого в течение восьми секунд в графическом интерфейсе ничего отображаться не будет, пока дочерняя программа не завершит работу. При наличии ключа графический интерфейс будет получать и отображать строки по мере их вывода, каждые две секунды.

Наконец, такой программный код, не использующий сокеты, не требующий вносить изменения в оригинальную программу и не блокирующий графический интерфейс, можно было бы использовать для отображения вывода программ командной строки в графическом интерфейсе. Конечно, во многих случаях может оказаться слишком сложным добавлять графический интерфейс таким способом, и для вас может оказаться проще превратить свой сценарий в традиционную программу с графическим интерфейсом, у которой есть главное окно и цикл событий. Кроме того, графические интерфейсы, которые мы реализовали в этом разделе, мы обязали просто отображать вывод другой программы, тогда как на практике от графического интерфейса может требоваться нечто большее. Однако для многих программ отделение представления от реализации, которое обеспечивает модель графического интерфейса, порождающего дочернюю программу командной строки, имеет свои преимущества – обе части приложения понять будет намного проще, если они не будут смешиваться.

Сокеты мы будем подробно рассматривать в следующей части книги, поэтому данное обсуждение следует рассматривать, как предварительное знакомство. Как мы увидим далее, все станет еще более интересным, как только мы начнем комбинировать графические интерфейсы, потоки выполнения и сокеты.

В следующей главе мы закончим обсуждение тем, касающихся исключительно графического интерфейса, где рассмотрим применение уже знакомых нам виджетов и приемов для реализации более практичных программ. Но перед этим в следующем разделе мы познакомимся с некоторыми крупными примерами графических интерфейсов, рассмотрев сценарии, которые запускают их автоматически и могут служить образцами, демонстрирующими возможности языка Python и библиотеки tkinter.

Запускающие программы PyDemos и PyGadgets

В завершение главы исследуем реализацию двух графических интерфейсов, с помощью которых запускаются основные примеры для этой книги. Следующие два графических интерфейса, PyDemos и PyGadgets, служат для запуска других программ с графическим интерфейсом. На самом деле мы подошли к концу истории о программах, запускающих демонстрационные примеры, – обе программы, представленные здесь,

взаимодействуют с модулями, с которыми мы встречались ранее, во второй части книги:

launchmodes.py

Запускает независимые программы Python переносимым образом.

Launcher.py

Отыскивает программы и в конечном итоге запускает обе программы, PyDemos и PyGadgets, при использовании самонастраивающихся сценариев верхнего уровня.

LaunchBrowser.py

Запускает веб-броузеры переносимым способом, открывая в них локальные или удаленные страницы.

Реализацию этих модулей вы найдете во второй части книги (особенно в главах 5 и 6). Представленные здесь программы добавляют компоненты графического интерфейса в систему запуска программ – они создают простые в использовании кнопки, нажатием которых можно запустить большинство крупных примеров, содержащихся в книге.

Кроме того, оба эти сценария предполагают, что при запуске текущим рабочим каталогом будет каталог, где они находятся (в них жестко определены пути к другим программам относительно этого каталога). Щелкните на их именах в проводнике по файловой системе или запустите из командной строки, выполнив команду `cd` для перехода в корневой каталог примеров *PP4E*. В этих сценариях можно было бы реализовать поддержку запуска и из других каталогов, путем использования значений переменных окружения для получения путей к сценариям, но в действительности они предназначены только для запуска из корневого каталога *PP4E*.

Поскольку эти сценарии запуска демонстрационных примеров являются достаточно длинными программами, в интересах экономии места на страницах книги будут приводиться только наиболее интересные их фрагменты. Полный программный код вы найдете в пакете с примерами.

Панель запуска PyDemos

Сценарий PyDemos создает панель с кнопками, которые запускают программы в демонстрационном режиме – не для повседневного применения. Я использую PyDemos, когда мне необходимо показать программы Python, – гораздо проще нажимать на кнопки, чем набирать командные строки или искать сценарии с помощью проводника по файловой системе.

Вы можете использовать PyDemos (и PyGadgets) для запуска и опробования примеров, представленных в этой книге, – все кнопки в этом графическом интерфейсе представляют примеры, с которыми мы познакомимся в последующих главах. Однако если вы соберетесь использовать

сценарии *Launch_PyDemos* и *Launch_PyGadgets_bar*, находящиеся в корневом каталоге с примерами, не забудьте включить в переменную окружения PYTHONPATH путь к каталогу *PP4E* – они не предусматривают автоматическую настройку вашей системы или путей поиска модулей.

Чтобы пользоваться этой панелью запуска было еще легче, перетащите ее на рабочий стол Windows, создав ярлык, на котором можно щелкнуть мышью (нечто подобное можно проделать и на других системах). Так как в этом сценарии жестко определены команды для запуска программ, находящихся в других подкаталогах в дереве примеров, он также полезен как предметный указатель к главным примерам из книги. На рис. 10.17 показано, как выглядит интерфейс сценария PyDemos при выполнении в Windows, наряду с несколькими демонстрационными программами, которые он запускает; PyDemos – это вертикальная панель с кнопками. В Linux он выглядит несколько иначе, но действует так же.

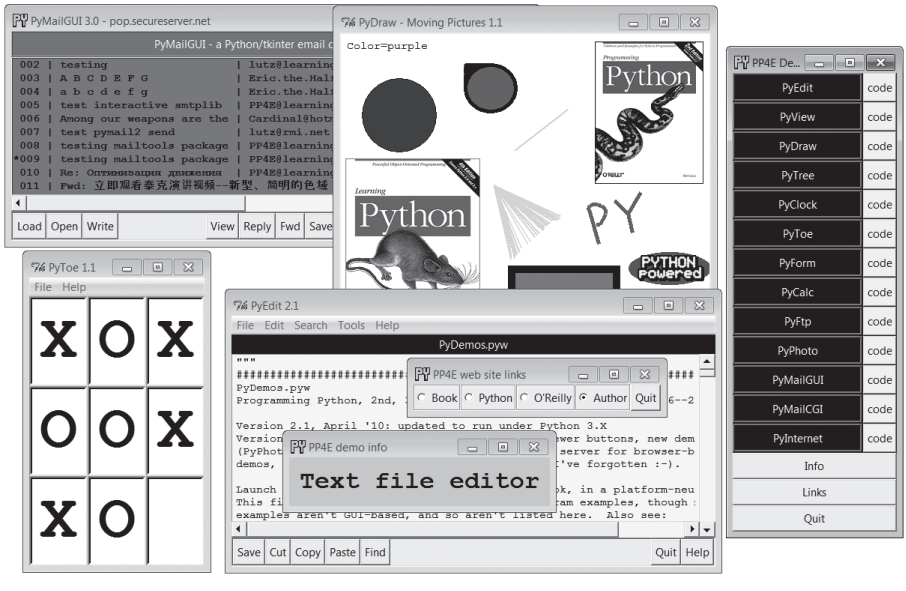


Рис. 10.17. *PyDemos с несколькими демонстрационными программами*

Исходный программный код, с помощью которого создается такая картина, приводится в примере 10.30 (его начало может несколько отличаться от того, что изображено на рис. 10.17, из-за мелких изменений, которые разработчики так любят вносить в последний момент). Сценарий PyDemos не содержит ничего особенного с точки зрения программирования графических интерфейсов, поэтому большая его часть не вошла в листинг – полную реализацию вы найдете в пакете с примерами.

В двух словах, функция demoButton в нем просто прикрепляет к главному окну новую кнопку, готовую при нажатии запустить программу

на языке Python. Для запуска программ сценарий PyDemos вызывает экземпляр объекта launchmodes.PortableLauncher, с которым мы познакомились в конце главы 5, – поскольку здесь он выступает в роли обработчика tkinter, для запуска программы используется операция вызова функции.

Как показано на рис. 10.17, сценарий PyDemos создает также два всплывающих окна, когда нажимаются кнопки в нижней части главного окна, – окно Info содержит краткое описание последней запущенной демонстрационной программы, а окно Links содержит переключатели, нажатие которых открывает связанные с книгой сайты в локальном веб-броузере:

- Всплывающее окно Info отображает 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 на кнопке Info, чтобы увидеть его или спрятать).
- Переключатели всплывающего окна Links своим поведением напоминают гиперссылки на веб-странице, но этот графический интерфейс на является броузером: при щелчке на них, с помощью сценария LaunchBrowser, упоминавшегося во второй части книги, отыскивается и запускается веб-броузер, подключающийся к соответствующему сайту при наличии соединения с Интернетом. Этот модуль в свою очередь использует современный модуль webbrowser из стандартной библиотеки Python.
- Чтобы ко всем окнам этого сценария привязать ярлык с синими буквами «PY» вместо стандартных красных букв «Tk», используется модуль windows, написанный нами ранее в этой главе.

В графическом интерфейсе сценария PyDemos также присутствуют кнопки code, расположенные правее кнопок с именами демонстрационных программ. Щелчок на этих кнопках открывает файлы с исходными текстами соответствующих примеров. Файлы открываются в текстовом редакторе PyEdit, с которым мы встретимся в главе 11. На рис. 10.18 изображены некоторые из окон с исходными текстами с несколько измененными размерами.

Для примеров, демонстрирующих работу с Интернетом, которые запускаются последними двумя кнопками на панели, выполняется попытка запустить локальный веб-сервер, обеспечивающий работу демонстрационных программ, не показанных здесь (мы встретимся с сервером в главе 15). В этом издании веб-серверы запускаются, только когда впервые выполняется щелчок на кнопке того или иного примера, демонстрирующего работу с Интернетом (а не при запуске PyDemos). При запуске веб-сервера в Windows открывается окно консоли, в которое выводятся сообщения о состоянии сервера.

PyDemos работает в Windows, Mac и в Linux в основном благодаря присущей переносимости Python и tkinter. Дополнительные подробности можно найти в исходных текстах, частично представленных в примере 10.30.

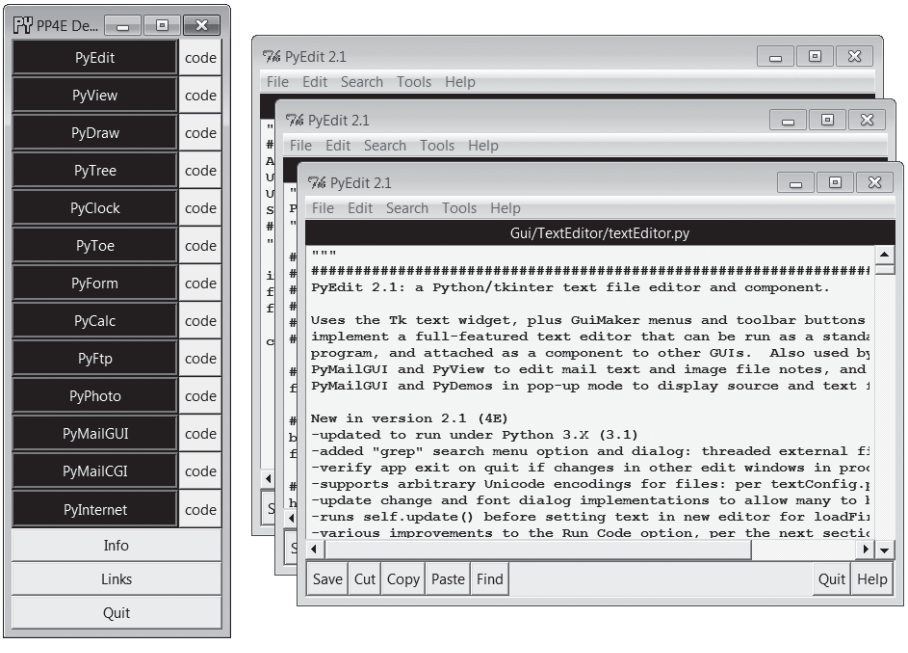


Рис. 10.18. *Сценарий PyDemos с окнами «code» для отображения исходных текстов*

Пример 10.30. PP4E\PyDemos.pyw (external)

```
"""
###############################################################################
PyDemos.pyw
Программирование на Python, 2, 3 и 4 издания (PP4E), 2001--2006--2010

Версия 2.1 (4E), апрель, 2010: добавлена возможность выполнения под управлением
Python 3.X и запуск локальных веб-серверов при первой попытке запустить пример,
демонстрирующий работу с Интернетом.

Версия 2.0 (3E), март, 2006: добавлены кнопки просмотра исходных текстов
примеров; добавлены новые демонстрационные программы (PyPhoto, PyMailGUI);
предусмотрен запуск локальных веб-серверов для демонстрационных примеров,
использующих веб-броузер; добавлены ярлыки окон; и, наверное, еще что-то,
о чем я забыл.

Запускает основные примеры графических интерфейсов Python+Tk из книги
независимым от платформы способом. Этот файл может также служить предметным
```

```
указателем к основным примерам программ, хотя многие примеры в книге не имеют
графического интерфейса и потому здесь не перечислены. Смотрите также:

- PyGadgets.py, более простой сценарий запуска программ в недемонстрационном
режиме, который можно использовать для повседневной работы
- PyGadgets_bar.pyw, создает панель с кнопками для запуска всех программ
PyGadgets по отдельности, а не всех сразу
- Launcher.py позволяет запускать программы без настройки окружения --
отыскивает Python, устанавливает PYTHONPATH и так далее.
- Launch_*.pyw, запускает PyDemos и PyGadgets с помощью Launcher.py --
попробуйте запустить их для беглого знакомства
- LaunchBrowser.pyw, открывает веб-страницы примеров в веб-броузере,
обнаруживаемом автоматически
- README-PP4E.txt, общая информация о примерах

ВНИМАНИЕ: эта программа пытается автоматически запускать локальный веб-сервер
и веб-броузер для демонстрационных примеров работы с Интернетом, но не завершает
работу сервера.
###############################################################################
“””
```

...часть программного кода опущена: смотрите файлы в дереве примеров...

```
###############################################################################
# начало создания главных окон графического интерфейса
###############################################################################

from PP4E.Gui.Tools.windows import MainWindow  # Tk с ярлыком, заголовком,
                                               # кнопкой закрытия
from PP4E.Gui.Tools.windows import PopupWindow # То же, но Toplevel,
                                               # отличается действием
                                               # кнопки закрытия
Root = MainWindow('PP4E Demos 2.1')

# создать окно сообщений
Stat = PopupWindow('PP4E demo info')
Stat.protocol('WM_DELETE_WINDOW', lambda:0)    # игнорировать событие

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 обработчиков
###############################################################################

from PP4E.Gui.TextEditor.textEditor import TextEditorMainPopup

# класс механизма запуска демонстрационных программ
class Launcher(launchmodes.PortableLauncher): # использовать имеющийся класс
```

```
    def announce(self, text):                # настроить метку в интерфейсе
        Info.config(text=text)

def viewer(sources):
    for filename in sources:
        TextEditorMainPopup(Root, filename,     # как всплывающее окно
                            loadEncode='utf-8') # иначе PyEdit может выводить
                                                # запросы для каждого!
def demoButton(name, what, doit, code):
    """
    добавляет кнопки, которые выполняют команды doit и открывают все
    файлы в списке code; кнопка doit сохраняет информацию в объекте, а кнопка
    code – в объемлющей области видимости;
    """
    rowfrm = Frame(Root)
    rowfrm.pack(side=TOP, expand=YES, fill=BOTH)

    b = Button(rowfrm, bg='navy', fg='white', relief=RIDGE, border=4)
    b.config(text=name, width=20, command=Launcher(what, doit))
    b.pack(side=LEFT, expand=YES, fill=BOTH)

    b = Button(rowfrm, bg='beige', fg='navy')
    b.config(text='code', command=(lambda: viewer(code)))
    b.pack(side=LEFT, fill=BOTH)

##############################################################################
# демонстрационные программы с графическим интерфейсом tkinter - некоторые
# используют сетевые соединения
##############################################################################

demoButton(name='PyEdit',
           what='Text file editor',                          # редактировать
           doit='Gui/TextEditor/textEditor.py PyDemos.pyw', # предполагается
           code=['launchmodes.py',                           # в тек. раб. кат.
                 'Tools/find.py',
                 'Gui/Tour/scrolledlist.py', # вывести в PyEdit
                 'Gui/ShellGui/formrows.py', # последний = верхний в стопке
                 'Gui/Tools/guimaker.py',
                 'Gui/TextEditor/textConfig.py',
                 'Gui/TextEditor/textEditor.py'])

demoButton(name='PyView',
           what='Image slideshow, plus note editor',
           doit='Gui/SlideShow/slideShowPlus.py Gui/gifs',
           code=['Gui/Texteditor/textEditor.py',
                 'Gui/SlideShow/slideShow.py',
                 'Gui/SlideShow/slideShowPlus.py'])
```

...часть программного кода опущена: смотрите файлы в дереве примеров...

```
##############################################################################
# переключение шрифта в окне Info 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: refreshMe(info, ncall+1)) )

##############################################################################
# показать/спрятать окно Info в случае щелчка на кнопке Info
##############################################################################

Stat.iconify()
def onInfo():
    if Stat.state() == 'iconic':
        Stat.deiconify()
    else:
        Stat.iconify() # было 'normal'

##############################################################################
# конец создания графического интерфейса, запуск цикла события
##############################################################################

def onLinks():
```

...часть программного кода опущена: смотрите файлы в дереве примеров...

```
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) # запустить переключение шрифтов в окне Info
Root.mainloop()
```

Панель запуска PyGadgets

Сценарий PyGadgets запускает часть тех же программ, что и PyDemos, но для практического использования, а не как кратковременные демонстрации. Оба сценария отображают панель с кнопками и запускают программы с помощью модуля `launchmodes`, но сценарий PyGadgets немного проще, потому что его задача более узкая. Кроме того, сценарий PyGadgets поддерживает два режима запуска – он может сразу запустить одновременно все программы из списка или вывести графический интерфейс для запуска каждой программы отдельно. На рис. 10.19 изображен графический интерфейс в виде панели с кнопками для запуска программ по отдельности. Сценарии PyGadgets и PyDemos могут выполняться одновременно, и оба позволяют изменять размеры окна (попробуйте сами, чтобы увидеть, как это делается).

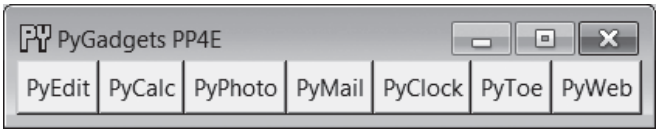


Рис. 10.19. *Панель запуска PyGadgets*

Из-за этих различий построение графического интерфейса в сценарии PyGadgets в большей мере основывается на данных: он сохраняет имена программ в списке и просматривает его при необходимости, а не проходит по последовательности заранее запрограммированных вызовов функции `demoButton`. Например, набор кнопок в панели запуска на рис. 10.19 целиком зависит от содержимого списка программ.

Программный код этого графического интерфейса приводится в примере 10.31. Его объем невелик, потому что опирается на использование других модулей, которые мы написали ранее, и осуществляющих большую часть его действий: `launchmodes` – для запуска программ, `LaunchBrowser` – для запуска веб-броузера, `windows` – для переопределения ярлыков и реализации операции завершения. На рабочем столе моего компьютера я создал ярлык для PyGadgets, и его окно практически всегда открыто у меня. С его помощью я легко получаю доступ к повседневно используемым инструментам – текстовым редакторам, калькуляторам, электронной почте, средствам обработки изображений 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 команд, запускающих программы, или измените список `mytools` вызываемых программ, который находится ближе к концу файла. В конце концов, это Python.

Пример 10.31. PP4E\PyGadgets.py

```
"""
###############################################################################
Запускает различные примеры; запускайте сценарий при загрузке системы, чтобы
сделать их постоянно доступными.
Этот файл предназначен для запуска программ, действительно необходимых в работе;
для запуска демонстрационных программ Python/Tk и получения дополнительных
сведений о параметрах запуска программ обращайтесь к сценарию PyDemos. Замечание
о работе в Windows: это файл с расширением '.py', поэтому при его запуске щелчком
мыши выводится окно консоли, которое используется для вывода начального сообщения
(включая 10-секундную паузу, чтобы обеспечить его видимость, пока запускаются
приложения). Чтобы избежать вывода окна консоли, запускайте сценарий с помощью
программы 'pythonw' (а не 'python'), используйте расширение '.pyw', в свойствах
ярлыка в Windows выберите значение 'Свернутое в значок' ('run minimized') в поле
'Окно' ('Window') или запускайте файл из другой программы (см. PyDemos).
###############################################################################
"""
```

```
import sys, time, os, time
from tkinter import *
from launchmodes import PortableLauncher # повторное использ. класса запуска
from Gui.Tools.windows import MainWindow # повторное использ. оконных
                                         # инструментов: ярлык, обработчик
                                         # закрытия окна
def runImmediate(mytools):
    """
    немедленный запуск программ
    """
    print('Starting Python/Tk gadgets...')    # вывод в stdout (временный)
    for (name, commandLine) in mytools:
        PortableLauncher(name, commandLine)() # сразу вызвать для запуска
    print('One moment please...')
    if sys.platform[:3] == 'win':             # windows: закрыть консоль через
        for i in range(10):                   # 10 секунд
            time.sleep(1); print('.' * 5 * (i+1))

def runLauncher(mytools):
    """
    создать простую панель запуска для использования в дальнейшем
    """
    root = MainWindow('PyGadgets PP4E')       # или root = Tk()
    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.py'),
    ('PyCalc',  'Lang/Calculator/calculator.py'),
    ('PyPhoto', 'Gui/PIL/pyphoto1.py Gui/PIL/images'),
    ('PyMail',  'Internet/Email/PyMailGui/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'),
    ('PyWeb', 'LaunchBrowser.pyw'
                        ' -live index.html learning-python.com')]
                       #' -live PyInternetDemos.html localhost:80')]
                       #' -file')] # PyInternetDemos предполагает, что
                                  # локальный веб-сервер уже запущен
if __name__ == '__main__':
    prestart, toolbar = True, False
    if prestart:
        runImmediate(mytools)
```

```
if toolbar:
    runLauncher(mytools)
```

По умолчанию сценарий PyGadgets сразу запускает все программы из списка. Чтобы запустить PyGadgets в режиме панели, в примере 10.32 импортируется и вызывается соответствующая функция с импортированным списком программ. Так как этот файл имеет расширение *.pyw*, на экране появится только графический интерфейс панели запуска – окно консоли открываться не будет. Это отлично подходит для повседневного использования, но не годится для случаев, когда желательно просматривать сообщения об ошибках (используйте расширение *.py*).

Пример 10.32. PP4E\PyGadgets_bar.pyw

```
"""
запускает только панель инструментов PyGadgets - ни одна другая программа при
этом не запускается; расширение файла предотвращает появление окна консоли
в Windows: используйте расширение '.py', чтобы видеть сообщения, выводимые
в консоль;
"""

import PyGadgets
PyGadgets.runLauncher(PyGadgets.mytools)
```

Этот сценарий – тот самый файл, на который ссылается ярлык на моем рабочем столе: я предпочитаю запускать приложения по мере необходимости. Создать ярлык и тем самым упростить возможность запуска можно на многих платформах. 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* (*Statrup*), а в Unix и в Unix-подобных системах можно автоматически запускать этот сценарий из командной строки в сценариях запуска, после запуска XWindow.

Каким бы способом ни был запущен сценарий PyGadgets – щелчком на ярлыке или на имени файла в проводнике по файловой системе, с помощью командной строки или иным образом, – появляется панель запуска, показанная в центре рис. 10.20.

Конечно, основное назначение сценария PyGadgets состоит в том, чтобы запускать другие программы. При нажатии на кнопки запускаются программы, показанные на рис. 10.20, и если вы хотите узнать о них больше, переверните страницу и перейдите к следующей главе.

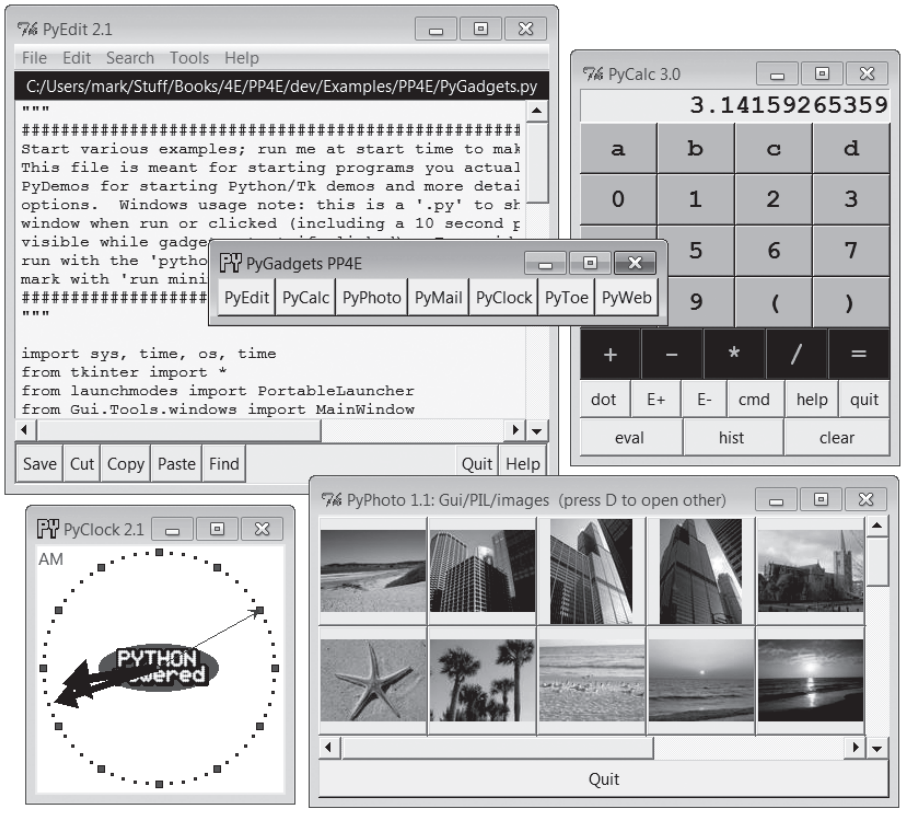


Рис. 10.20. *Панель запуска PyGadgets с несколькими запущенными приложениями*

11

Примеры законченных программ с графическим интерфейсом

«Python, открытое программное обеспечение и Camaro[1]»

Эта глава завершает тему создания графических интерфейсов с помощью языка Python и его стандартной библиотеки tkinter, представляя ряд практических программ с графическим интерфейсом. В четырех предшествующих главах мы освоили основы программирования с использованием библиотеки tkinter. Познакомились с базовым набором *виджетов* – классов Python, которые генерируют графические элементы управления на экране компьютера и могут реагировать на события, вызываемые пользователем. Кроме того, мы также изучили множество дополнительных приемов программирования графических интерфейсов, включая анимацию, перенаправление потоков ввода-вывода с применением сокетов и каналов и поддержку многопоточной модели выполнения. В данной главе мы сконцентрируемся на объединении этих виджетов и приемов для создания более полезных графических интерфейсов. Нами будут изучены:

PyEdit

Программа текстового редактора

PyPhoto

Программа просмотра миниатюр графических изображений

[1] Имеется в виду модель автомобиля «Chevrolet Camaro», пользующаяся славой надежного и неприхотливого автомобиля. – *Прим. перев.*

PyView

Программа просмотра графических изображений в режиме слайд-шоу

PyDraw

Графический редактор

PyClock

Графические часы

PyToe

Простая игра «крестики-нолики» в качестве развлечения[1]

Как и в главе 6, я выбрал примеры для этой главы из собственной библиотеки программ на языке Python, которыми я действительно пользуюсь. Например, текстовый редактор и часы, 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 в Windows и Linux и должны работать так же в Mac OS.

Так как эти сценарии написаны исключительно на языке Python, их дальнейшее развитие целиком зависит от пользователей – освоившись с интерфейсами tkinter, не составит труда изменить или улучшить поведение таких программ редактированием их программного кода. Некоторые из этих примеров аналогичны коммерческим программам (например, PyEdit напоминает Блокнот в Windows), однако переносимость сценариев Python и почти полное отсутствие препятствий к их дальнейшему улучшению дают им явное преимущество.

Примеры в других главах

Далее в этой книге мы встретим другие программы с графическим интерфейсом на базе tkinter, представляющие удобные инструменты для конкретных прикладных областей. В частности, в последующих главах появятся такие крупные примеры, как:

PyMailGUI

Клиент электронной почты (глава 14)

---

[1] Названия всех крупных примеров этой книги начинаются с приставки «Ру». Это – соглашение, принятое в мире Python. Если покопаться на сайте *http://www.python.org*, можно найти другое свободно распространяемое программное обеспечение, следующее этой схеме именования: PyOpenGL (интерфейс к графической библиотеке OpenGL для языка Python), PyGame (набор инструментов для разработки игровых программ на языке Python) и многие другие. Я не знаю, с кого это началось, но эта схема оказалась достаточно «Пи-кантным» способом рекламы языка программирования для всего мира программного обеспечения с открытыми исходными текстами. Если Питонист слишком прямолинеен – это не Питонист!

PyForm

(Внешний пример) Средство просмотра таблиц хранимых объектов (глава 17)

PyTree

(Внешний пример) Средство просмотра древовидных структур данных (главы 18 и 19)

PyCalc

Настраиваемый виджет калькулятора (глава 19)

Менее крупные примеры, включая клиента FTP и инструменты передачи файлов, будут также представлены в части главы, посвященной созданию сценариев для Интернета. Большей частью этих программ я постоянно пользуюсь. Так как библиотеки разработки графического интерфейса являются инструментами общего назначения, найдется совсем немного областей, которые не выиграли бы от простого в использовании, простого в программировании и хорошо переносимого интерфейса, реализованного на языке Python с применением tkinter.

Помимо примеров, представленных в этой книге, для Python существует множество высокоуровневых инструментальных средств создания графических интерфейсов, таких как Pmw, Tix и ttk, упомянутые в главе 7. Некоторые из этих систем опираются на библиотеку tkinter и предоставляют составные компоненты, такие как виджеты с закладками, деревья и всплывающие подсказки.

В следующей части книги мы также исследуем программы, которые строят интерфейсы пользователя в веб-броузерах, без использования tkinter, – совершенно иной подход к созданию пользовательских интерфейсов. Несмотря на то, что исторически веб-интерфейсы имеют более ограниченные возможности и в их работе часто наблюдаются задержки, связанные с передачей данных по сети, тем не менее при объединении с инструментами разработки полнофункциональных интернет-приложений (Rich Internet Application, RIA), упоминавшихся в начале главы 7, современные веб-интерфейсы по своим возможностям приближаются к традиционным графическим интерфейсам, хотя и за счет существенной сложности программного обеспечения.

Тем не менее для создания высокоинтерактивных и нетривиальных интерфейсов автономные, настольные графические интерфейсы, предлагаемые библиотекой tkinter, окажутся незаменимым инструментом практически для всех приложений на языке Python. Программы, демонстрируемые в этой главе, позволяют увидеть, каких высот можно достичь с помощью Python и tkinter.

Стратегия данной главы

Как и все главы этой книги, посвященные исследованию конкретных случаев, данная глава в значительной мере является «обучением на

примере» – текст большинства программ приведен с минимумом подробностей. По ходу дела я буду отмечать важные точки и новые возможности tkinter, представляемые каждым примером, но помимо этого я полагаюсь на то, что вы самостоятельно изучите детали по приведенным листингам и комментариям. Легкость чтения Python становится существенным достоинством для программистов (и авторов книг), особенно когда сложность программ достигает такого уровня, как в этой главе.

Исходные тексты всех примеров графических интерфейсов, упоминаемых в этой книге, доступны в пакете с примерами, как описывалось в предисловии. Поскольку ранее я уже описывал функции и методы, используемые этими сценариями, в этом разделе будут приводиться в основном снимки с экрана и листинги программ, сопровождаемые кратким описанием некоторых из наиболее важных аспектов этих программ. Иными словами, этот раздел предназначен для самостоятельного изучения: читайте исходные тексты, запускайте примеры на своем компьютере и обращайтесь к предыдущим главам за дополнительными подробностями. Некоторые из этих программ могут также сопровождать альтернативные или экспериментальные реализации в пакете с примерами, не перечисленные здесь, – ищите дополнительные примеры в дереве каталогов с примерами.

Наконец, я хочу напомнить, что все перечисленные выше крупные программы можно запускать из панелей запуска PyDemos и PyGadgets, с которыми мы встретились в конце главы 10. Я попытаюсь передать их поведение на снимках экранов, которые будут приведены здесь, но графические интерфейсы по своей природе являются системами, управляемыми событиями, и чтобы опробовать характер их взаимодействия с пользователем, лучше реального запуска примера ничего не придумаешь. Поэтому панели запуска фактически являются дополнением к материалу данной главы. Они могут выполняться на большинстве платформ и обеспечивают легкость запуска (ищите подсказки в файле *README-PP4E.txt*). Запускайте их и сразу начинайте щелкать мышью, если еще не сделали этого.

Открытое программное обеспечение и Camaro

Некоторые из программ с графическим интерфейсом, представленные в этой главе и в оставшейся части книги, являются аналогами утилит, которые можно найти в наиболее распространенных операционных системах, таких как Windows. Например, мы рассмотрим калькуляторы, текстовые редакторы, средства просмотра графических изображений, часы, клиенты электронной почты и другие.

Но, в отличие от большинства утилит, эти программы являются переносимыми – благодаря тому, что они написаны на языке Python с применением библиотеки tkinter, эти программы способны работать на всех основных платформах (Windows, Unix/Linux и Mac). Но самое важное, пожалуй, – они могут настраиваться под личные предпочтения, благодаря доступности исходных текстов, – вы можете изменять их внешний вид или функциональные возможности, просто дописав или изменив программный код на языке Python.

Приведу аналогию, чтобы подчеркнуть важность возможности что-то настраивать и переделывать под себя. Среди нас еще есть люди, которые помнят времена, когда считалось нормальным, если владелец автомобиля сам ухаживал за ним и ремонтировал его. Я с нежностью вспоминаю, как в годы моей юности мы с друзьями увлеченно копались под капотом Chevrolet Camaro 1970, ремонтируя и отлаживая его двигатель. Приложив совсем немного усилий, мы смогли увеличить его скорость, приемистость и придать его работе звучание, услаждавшее наш слух. Кроме того, поломка какого-то из наших старых автомобилей не была для нас концом света. Всегда оставался шанс самостоятельно починить его.

Сейчас все изменилось. С появлением электронных средств управления и дьявольски тесных моторных отсеков владельцы автомобилей стали предпочитать пользоваться услугами специалистов в любых, даже в самых простых случаях. В целом, автомобили перестали быть продуктом, доступным для самостоятельного ремонта. И если в моем новеньком, сверкающем экипаже случится поломка, я наверняка застряну на дороге, пока подготовленный специалист не найдет время, чтобы отбуксировать его и отремонтировать.

Я люблю сравнивать закрытую и открытую модели разработки программного обеспечения, оперируя теми же понятиями. Когда я использую программы корпорации Microsoft, такие как Notepad и Outlook, я ограничен возможностями, предусмотренными компанией-производителем, а также вынужден мириться со всеми ошибками, которые могут скрываться в этих программах. А в случае с такими программными инструментами, как PyEdit и PyMailGUI, у меня сохраняется возможность «залезть под капот». Я могу добавлять новые особенности, настраивать систему и исправлять любые ошибки, какие будут обнаружены. И могу сделать это намного быстрее, чем Microsoft выпустит очередной набор исправлений или новую версию своего продукта. Я не завишу от компании, действующей в общем-то в своих интересах, если мне требуется поддержка или даже продолжение разработки инструментов, которыми я пользуюсь.

Конечно, я по-прежнему завишу от языка Python и от тех изменений, которые могут в него вноситься с течением времени (после обновления двух книг, по тысяче страниц с лишним в каждой, под Python 3.X, я с определенной уверенностью могу сказать, что эта зависимость не всегда является тривиальной). Однако наличие исходных текстов для всех программных инструментов, на которые вы полагаетесь, все равно можно считать мощной поддержкой и крупным преимуществом. А кроме того, открытая модель способствует повышению надежности, предоставляя возможность сообществам людей тестировать и развивать систему.

В конечном счете, открытое программное обеспечение и Python чаще всего ассоциируются со *свободой*, тогда как закрытое – с ценой. Последнее слово здесь остается за пользователями, а не за какой-то далекой компанией. Конечно, не каждый захочет возиться со своим автомобилем. Но, с другой стороны, программное обеспечение имеет свойство терпеть неудачу намного чаще, чем автомобили – ломаться, да и программирование на языке Python является менее грязной работой, чем работа автомеханика.

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 воплощены их разработчиками), и многие работали только на одной платформе. Представленная в этом разделе программа PyEdit более удачна в обоих отношениях: она реализует полноценный текстовый редактор с графическим интерфейсом в **1133** строках переносимого программного кода на языке Python, включая пробельные символы и комментарии, из которых 1088 строк содержатся в главном файле и 45 строк – в модуле с настройками (к моменту выхода этого издания книги; в будущем его размер может измениться). Несмотря на свой относительно скромный размер, редактор PyEdit оказался достаточно мощным и надежным, чтобы послужить основным инструментом для разработки 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, отмена и возврат ввода и так далее. Но в действительности PyEdit представляет собой нечто большее, чем просто текстовый редактор, – его можно использовать как программу и как библиотечный компонент, и он может использоваться в разных режимах:

Автономный режим

В качестве автономной программы текстового редактора, с возможностью передачи имени редактируемого файла в командной строке. В этом режиме PyEdit напоминает другие утилиты редактирования текста (например, Notepad в Windows), но, кроме того, предоставляет дополнительные возможности, такие как запуск редактируемой программы на языке Python, изменение шрифта и цвета, поиск во внешних файлах, многооконный интерфейс и так далее. Но особенно важно, что текстовый редактор PyEdit легко можно модифицировать и использовать в Windows, X Window и Macintosh, потому что он написан на языке Python.

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 от других. В этом и в следующем режимах PyEdit служит библиотечным объектом, используемым другими сценариями, а не готовым приложением. Например, приложение PyMailGUI, представленное в главе 14, использует PyEdit во всплывающем режиме для отображения вложений в электронные письма и простого текста, и оба приложения, PyMailGUI и PyDemos из предыдущей главы, отображают исходный программный код таким способом.

Встроенный режим

В качестве прикрепляемого компонента – виджета редактирования текста для других графических интерфейсов. Будучи прикрепленным, PyEdit использует меню, основанное на фрейме, и может отключать некоторые его пункты, не имеющие смысла во встроенном режиме. Например, программа PyView (рассматривается далее в этой главе) использует PyEdit во встроенном режиме в качестве редактора подписей к фотографиям, а PyMailGUI (в главе 14) прикрепляет его и бесплатно получает редактор текста электронных писем.

Может показаться, что такое поведение с разными режимами трудно реализовать, но на самом деле режимы PyEdit по большей части являются естественным побочным продуктом разработки графического интерфейса с использованием подхода, основанного на применении классов, рассматривавшегося на протяжении последних четырех глав.

Запуск PyEdit

Редактор PyEdit обладает массой возможностей, и лучший способ понять, как он действует, – поработать с ним самостоятельно. Его можно открыть, запустив главный файл *textEditor.py* или файлы *textEditor-*

NoConsole.pyw и *pyedit.pyw*, если желательно подавить появление окна консоли в Windows, или воспользовавшись панелями запуска демонстрационных программ PyDemos и PyGadgets, которые были описаны в конце главы 10 (сами запускающие программы находятся на верхнем уровне дерева каталогов примеров книги). Чтобы вы могли получить представление об интерфейсах PyEdit, на рис. 11.1 изображено главное окно программы – как оно выглядит по умолчанию в Windows 7 после открытия файла с исходными текстами PyEdit.

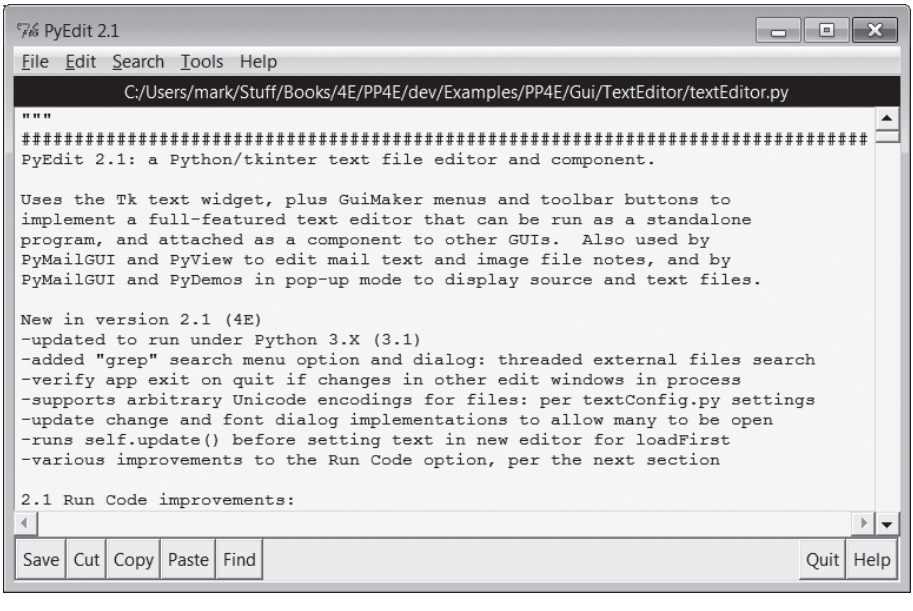


Рис. 11.1. *Главное окно редактора PyEdit с программным кодом самого редактора*

Главную часть этого окна составляет виджет `Text`, и если вы прочли его описание в главе 9, вам должны быть знакомы операции редактирования текста, выполняемые PyEdit. В нем используются метки, теги и индексы, и реализованы операции удаления и вставки через системный буфер обмена, который позволяет вставлять скопированные данные, даже после того как приложение-источник было закрыто. Для поддержки произвольного перемещения по содержимому файлов с виджетом `Text` взаимно связаны вертикальная и горизонтальная полосы прокрутки.

Меню и панели инструментов

Меню и панели инструментов редактора PyEdit должны показаться вам знакомыми – он строит главное окно, используя минимальный объем программного кода, и обеспечивает действие соответствующих правил

обрезания и растягивания путем внедрения класса `GuiMaker`, с которым мы познакомились в предыдущей главе (пример 10.3). Панель инструментов внизу окна содержит кнопки для быстрого доступа к операциям, которыми я пользуюсь чаще всего; если ваши вкусы не совпадают с моими, просто измените список кнопок панели инструментов в исходном программном коде, чтобы в нем оказались кнопки, которые вам нужны (в конце концов, это Python).

Как обычно, в меню tkinter для быстрого вызова элементов меню можно использовать горячие клавиши – следует нажать Alt и все подчеркнутые клавиши на пути к нужному действию. Меню могут также отрываться по пунктирной линии и тем самым обеспечить быстрый доступ к пунктам меню в новом окне верхнего уровня (удобно, когда отсутствует панель инструментов с кнопками).

Диалоги

PyEdit выводит различные модальные и немодальные диалоги, стандартные и собственные. На рис. 11.2 изображены нестандартные, немодальные диалоги поиска с заменой, выбора шрифта и поиска во внешних файлах, а также стандартный диалог для вывода информации о файле (окончательные значения счетчиков в последних строках могут измениться, потому что я имею обыкновение изменять программный код и добавлять комментарии вплоть до окончания проекта).

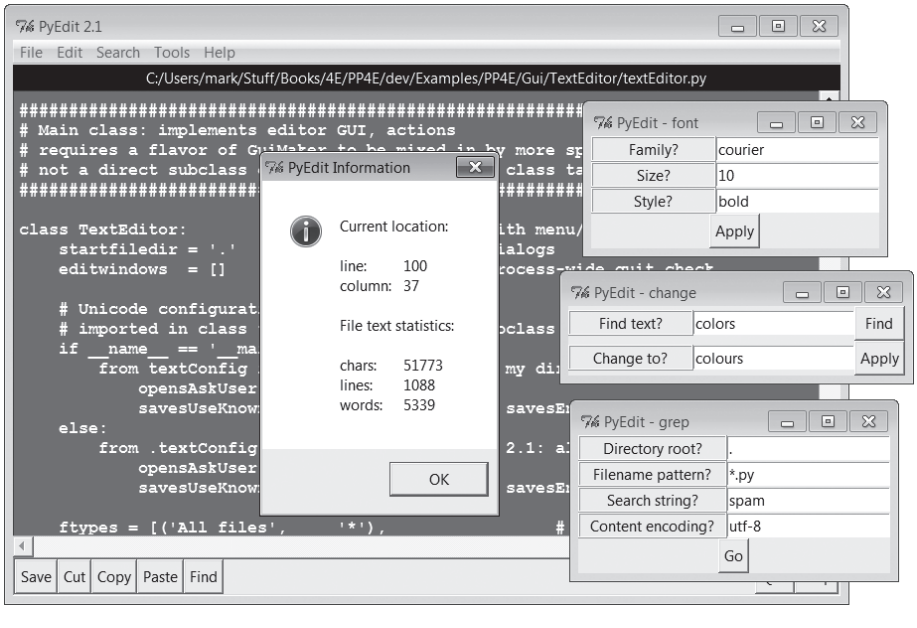


Рис. 11.2. *PyEdit с измененными цветами, шрифтом и некоторыми диалогами*

В главном окне на рис. 11.2 установлены новые цвета переднего плана и фона (с помощью стандартного диалога выбора цвета) и новый шрифт, который можно установить либо с помощью диалога выбора шрифта, либо из имеющегося в сценарии готового списка, который пользователи могут изменять в соответствии со своими предпочтениями (в конце концов, это Python). Другие операции, выполняемые с помощью панели инструментов и меню, обычно используют стандартные диалоги с некоторыми дополнениями. Например, при работе со стандартными диалогами открытия и сохранения файлов в PyEdit используются интерфейсы на основе объектов, которые запоминают каталог, выбиравшийся последним, и устраняют необходимость каждый раз заново переходить к нему.

Запуск программного кода

Одной из уникальных особенностей PyEdit является возможность запуска редактируемого в нем программного кода на языке Python. Это не так сложно, как может показаться. В Python имеются встроенные функции компиляции и выполнения строк программного кода, а также запуска программ, поэтому редактору PyEdit остается лишь выполнить вызовы нужных функций. В частности, на языке Python легко можно написать простенький интерпретатор Python, как показано ниже (если вы захотите поэкспериментировать с ним, найдите файл *simpleShell.py* в каталоге с реализацией PyEdit), хотя осуществить обработку многострочных инструкций и отображение результатов выражений несколько сложнее.

```
# читает и выполняет строки с инструкциями на языке Python: подобно тому,
# как действует пункт 'Run Code' в меню PyEdit
namespace = {}
while True:
    try:
        line = input('>>> ')   # только однострочные инструкции
    except EOFError:
        break
    else:
        exec(line, namespace)  # или eval() и вывод результата
```

В зависимости от предпочтений пользователя редактор PyEdit или делает что-то подобное этому, чтобы выполнять программный код, выбираемый из текстового виджета, или использует модуль `launchmodes`, который мы написали в конце главы 5, чтобы запустить 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 реализации смотрите в методе `onRunCode` или просто отредактируйте и выполните свой собственный программный код на языке Python. Когда выполняется только выбранная строка программного кода, вы сможете наблюдать результаты в окне консоли редактора PyEdit. Как уже говорилось в примечании о функциях `eval`

и exec в главе 9, этим функциям следует передавать программный код только из проверенных источников – он получает доступ ко всему, что доступно процессу интерпретатора Python.

Несколько окон

Редактор PyEdit способен выводить не только множество окон специального назначения, он также позволяет одновременно открывать несколько окон редактирования – в пределах одного процесса или за счет запуска независимых экземпляров программы. Для иллюстрации на рис. 11.3 показаны три независимо выполняющиеся экземпляра PyEdit с различными размерами, цветовыми схемами и шрифтами. Поскольку все они были запущены как независимые программы, закрытие одной из них не оказывает влияния на другие. На этом рисунке внизу видны также оторванные меню PyEdit и всплывающее окно диалога справки справа. Фоном окон редактирования служат оттенки зеленого, красного и голубого цветов – для установки желаемого цвета выберите в меню Tools элемент Pick.

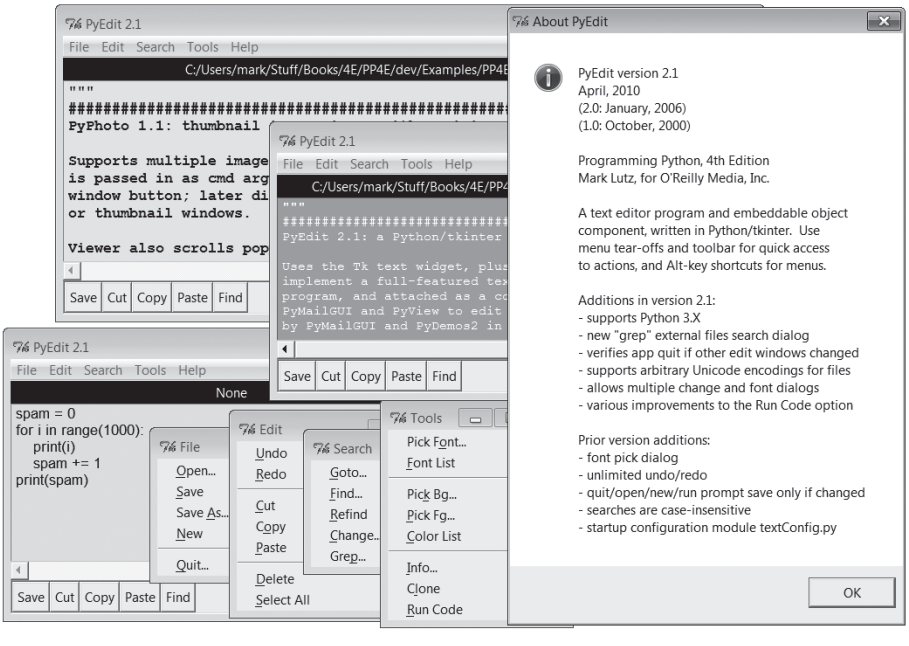


Рис. 11.3. *Несколько окон PyEdit, открытых одновременно*

Так как во всех этих трех сеансах PyEdit редактируется программный код на языке Python, их содержимое можно выполнить, выбрав пункт Run Code раскрывающегося меню Tools. Программный код из файлов выполняется независимо – стандартные потоки ввода-вывода программного кода, выполняемого не из файла (например, полученные из самого

текстового виджета), отображаются в окно консоли сеанса PyEdit. Это никоим образом нельзя рассматривать, как IDE (интегрированную среду разработки), – я добавил эту возможность только потому, что она показалась мне полезной. Очень удобно иметь возможность запускать редактируемый программный код, не разыскивая его в дереве каталогов.

Чтобы открыть несколько окон редактирования в пределах одного процесса, используйте пункт Clone в меню Tools, при выборе которого открывается новое пустое окно без уничтожения содержимого в другом окне. На рис. 11.4 показана ситуация, когда в одном процессе открыты два окна, наряду со всплывающими окнами, имеющими отношение к пункту Grep меню Search, о котором рассказывается в следующем разделе, – который позволяет произвести обход дерева каталогов в параллельных потоках выполнения, отобрать файлы с именами, соответствующими шаблону и содержащими искомую строку, и открыть их. На рис. 11.4 видно, что выбор пункта Grep меню выводит диалог ввода, список файлов, удовлетворяющих критериям поиска и новое окно PyEdit, открытое и позиционированное после двойного щелчка на файле в списке.

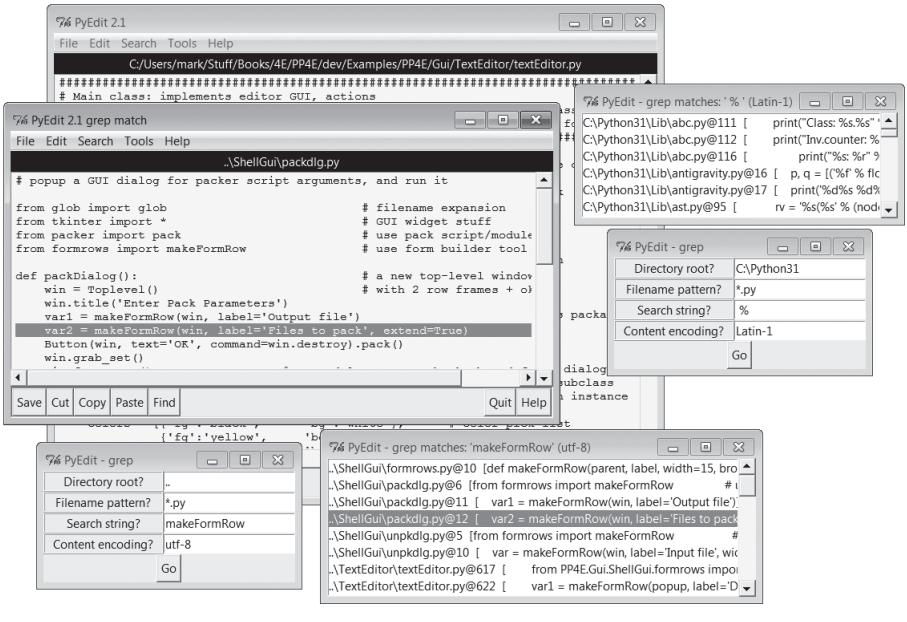


Рис. 11.4. *Множество окон PyEdit в рамках единственного процесса*

В процессе выполнения операций, предусмотренных пунктом Grep меню, на экране появляется еще одно окно, но при этом графический интерфейс остается полностью активным. Фактически вы можете снова выбрать пункт Grep и выполнить еще один поиск, пока другой еще не закончился. Обратите внимание, что диалог Grep также позволяет указывать кодировку символов, которая будет использоваться для декоди-

рования содержимого всех текстовых файлов, просматриваемых в процессе поиска. Подробнее о том, как это действует, я расскажу в следующем ниже разделе с описанием изменений, однако в большинстве случаев можно просто использовать системную кодировку по умолчанию.

Ради интереса попробуйте вызвать диалог Grep и выполнить в каталоге *C:\Python31* поиск всех файлов **.py*, содержащих строку %, чтобы получить представление о том, как часто используется это распространенное выражение форматирования строк в стандартной библиотеке Python 3.1. Разумеется, не все вхождения % будут иметь отношение к форматированию строк, но большинство из них – точно. Согласно сообщениям, выводимым в стандартный поток вывода перед завершением поиска, строка '%' (которая также соответствует целям для подстановки) встречается 6050 раз, а строка ' % ' (с пробелами вокруг знака процента, чтобы ограничить круг совпадений только оператором форматирования) встречается 3741 раз, включая 130 совпадений, обнаруженных в установленном расширении PIL, – не самый редко используемый инструмент языка! Ниже приводятся сообщения, которые были выведены в стандартный поток вывода в процессе поиска, – совпадения выводятся также в окно списка:

...ошибки могут отличаться от типа кодировки...

```
Unicode error in: C:\Python31\Lib\lib2to3\tests\data\different_encoding.py
Unicode error in: C:\Python31\Lib\test\test_doctest2.py
Unicode error in: C:\Python31\Lib\test\test_tokenize.py
Matches for % : 3741
```

Редактор PyEdit выводит дополнительные всплывающие окна – включая кратковременные диалоги Goto и Find, диалоги выбора цвета, диалоги ввода аргументов и выбора режимов для пункта Run Code меню, и диалоги, запрашивающие имена кодировок символов в файлах при выборе в меню пунктов Open и Save, если редактор настроен так, что он должен запрашивать эти данные (подробнее об этом рассказывается ниже). В интересах экономии места я оставляю исследование большинства других подобных черт поведения за вами.

Существенно обновленный в этом издании и поддерживающий настройки в соответствии с пользовательскими предпочтениями, редактор PyEdit может запрашивать имя кодировки символов при открытии файла, сохранять совершенно новый файл или выполнять операцию Save As. Например, на рис. 11.5 изображен момент, когда я открыл файл, содержащий символы китайского алфавита, и затем снова выбрал в меню пункт Open, чтобы открыть новый файл с текстом на русском языке. Диалог выбора имени кодировки, изображенный на рисунке, появляется сразу же после закрытия стандартного диалога выбора файла, а поле ввода в нем предварительно заполнено именем кодировки по умолчанию (которое определяется явно или берется из настроек платформы). В большинстве случаев можно использовать имя, предла-

гаемое по умолчанию, если только заранее не известно точно, что символы в файле представлены в другой кодировке.

Вообще говоря, редактор PyEdit поддерживает любые кодировки, которые поддерживаются языком Python и библиотекой tkinter. Текст, который можно видеть на рис. 11.5, например, содержал символы китайского алфавита в специфической кодировке (в кодировке «gb2321», в файле *email-part--gb2312*). В том же каталоге присутствует тот же текст в альтернативной кодировке UTF-8 (файл *email-part--gb2312--utf8*), который можно открывать в PyEdit и Notepad, используя кодировку по умолчанию, используемую в Windows. Но, чтобы открыть файл в специфической китайской кодировке и получить корректное отображение символов в PyEdit, требуется явно указать имя кодировки (содержимое этого файла абсолютно неправильно отображается в Notepad).

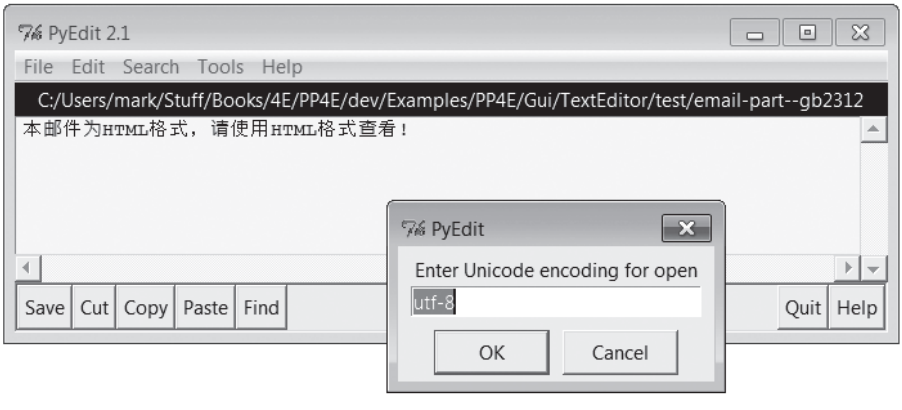


Рис. 11.5. *PyEdit отображает китайский текст и запрашивает имя кодировки при открытии файла*

После того как я ввел имя кодировки для выбранного файла в диалоге на рис. 11.5 («koi8-r» – для файла, выбранного в диалоге открытия), редактор PyEdit декодировал и отобразил его содержимое. На рис. 11.6 изображен момент, после того как файл был открыт и я выбрал в меню пункт Save As, – сразу после того как был закрыт диалог выбора файла, редактор вывел новый диалог ввода имени кодировки для нового файла, поле ввода в котором было предварительно заполнено именем кодировки, известным по последним операциям Open или Save. В соответствии с настройками операция Save повторно использует известную кодировку, но операция Save As всегда запрашивает кодировку, чтобы дать возможность указать ее явно для нового файла, прежде чем пытаться использовать умолчания. Подробнее об алгоритмах применения кодировок и интернационализации в PyEdit я буду рассказывать в следующем разделе, при обсуждении изменений в версии 2.1, а пока отмечу, что из-за того, что предпочтения пользователя не могут быть предугаданы, выбор среди алгоритмов поддерживается настройками.

Наконец, когда приходит время завершать работу, редактор PyEdit делает все возможное, чтобы не потерять несохраненные изменения. Когда в любом окне редактирования запрашивается выполнение операции завершения, PyEdit проверяет наличие несохраненных изменений и запрашивает подтверждение. Поскольку в одном и том же процессе может быть открыто несколько окон редактирования, когда операция завершения запрашивается в главном окне, PyEdit проверяет наличие изменений во всех остальных открытых окнах и запрашивает подтверждение, если хотя бы в одном из них будут обнаружены несохраненные изменения. В противном случае операция завершения будет выполнена без дополнительных вопросов. Попытка выполнить операцию завершения во всплывающем окне редактирования закроет только это окно, то есть никаких проверок между процессами выполняться не будет. При отсутствии изменений операция завершения просто закроет окна графического интерфейса и завершит программу. Другие операции проверяют наличие изменений похожим способом.

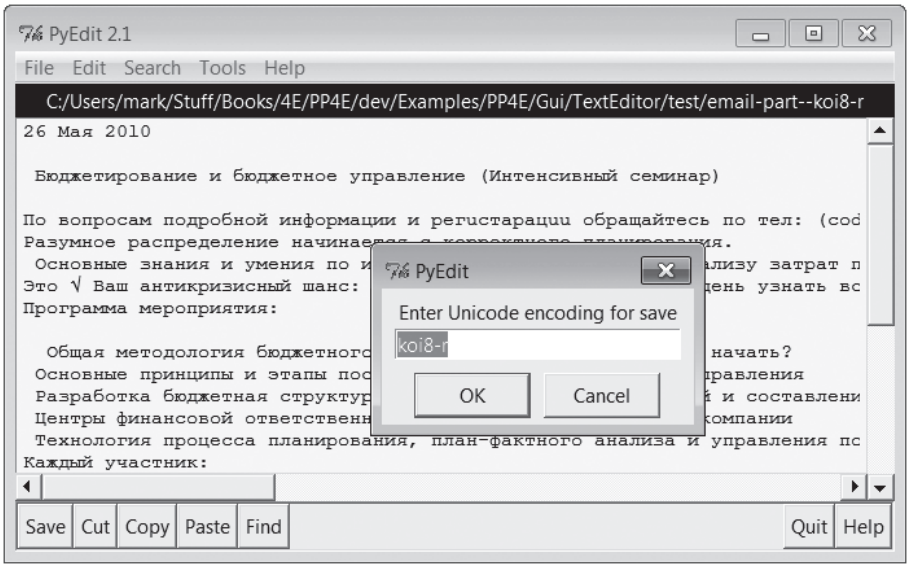


Рис. 11.6. *PyEdit отображает текст на русском языке и запрашивает кодировку для операции Save As*

Другие примеры и рисунки с изображением PyEdit в этой книге

Дополнительные рисунки с изображением PyEdit можно найти в описаниях следующих программ-клиентов:

- PyDemos в главе 10 использует PyEdit во всплывающем режиме для отображения файлов с исходными текстами.

- PyView далее в этой главе использует PyEdit во встроенном режиме для отображения примечаний к файлам изображений.
- PyMailGUI в главе 14 использует PyEdit для отображения сообщений электронной почты, текстовых вложений и исходных текстов.

Последнее приложение особенно интенсивно использует функциональные возможности PyEdit, а в главе с его описанием имеются рисунки, демонстрирующие возможности PyEdit по отображению текста Юникода с национальными наборами символов. При таком использовании текст либо извлекается из сообщений, либо загружается из временных файлов, кодировка которых определяется из заголовков сообщений электронной почты.

Изменения в версии PyEdit 2.0 (третье издание)

Я изменял этот пример в обоих, в третьем и четвертом, изданиях этой книги. Поскольку эта глава призвана отражать практические приемы программирования, а также потому, что этот пример демонстрирует процесс развития программного обеспечения с течением времени, этот и следующий разделы дают краткое описание основных изменений, выполненных за это время, чтобы помочь вам в изучении программного кода.

Поскольку текущая версия наследует все улучшения от предшествующих ей, начнем с дополнений, появившихся в предыдущей версии. В третьем издании редактор PyEdit был дополнен следующими возможностями:

- Простой диалог выбора шрифта
- Поддержка неограниченного количества отмен и возвратов операций редактирования
- Проверка наличия изменений в файле, когда его содержимое могло быть удалено или изменено
- Модуль для хранения настроек пользователя

Далее приводятся некоторые краткие примечания, касающиеся этих дополнений.

Диалог выбора шрифта

В третьем издании в редактор PyEdit был добавлен *диалог выбора шрифта* – простой немодальный диалог с тремя полями, куда можно ввести название семейства шрифта, размер и стиль, вместо того чтобы выбирать из предопределенного списка возможных вариантов. Хотя вы можете найти более сложные диалоги выбора шрифта на основе библиотеки tkinter, используемые в общедоступных приложениях и в реализации стандартной среды разработки IDLE на языке Python (как уже упоминалось ранее, среда IDLE сама по себе является программой, написанной на языке Python и использующей библиотеку tkinter).

Отмена, возврат и проверка наличия изменений

Еще одной новинкой в версии PyEdit для третьего издания стала поддержка неограниченного количества *отмен и возвратов* (undo/redo) операций редактирования, проверка наличия *изменений* перед завершением редактора, а также перед выполнением операций открытия, запуска и создания нового файла, чтобы при необходимости запросить сохранение этих изменений. Теперь запрос подтверждения на выход или перезапись файла выводится уже не каждый раз, а только если текст в окне редактора действительно изменился. Библиотека Tk версии 8.4 (или выше) предоставляет прикладной интерфейс, который упрощает реализацию обеих этих возможностей, – Tk сохраняет стеки отмены и возврата операций редактирования автоматически. Они включаются с помощью параметра `undo` настройки виджета `Text` и доступны с помощью методов `edit_undo` и `edit_redo`. Аналогично метод `edit_reset` очищает стеки (например, после открытия нового файла), а метод `edit_modified` проверяет или устанавливает признак наличия изменений в тексте.

Отмену вырезания и вставки текста из буфера обмена сразу после их выполнения реализовать совсем несложно (простой вставкой текста из буфера обмена или вырезанием вставленного и выделенного текста), но усовершенствованная поддержка операций отмены/возврата более полная и проще в использовании. Во втором издании книги реализация отмены была предложена в качестве самостоятельного упражнения, но она стала практически тривиальной благодаря новому прикладному интерфейсу библиотеки Tk.

Модуль с настройками

Для большего удобства версия PyEdit в третьем издании была дополнена возможностью определять начальные *параметры настройки* за счет присваивания значений переменным в модуле *textConfig.py*. Если поместить этот файл в путь поиска модулей, при импортировании или запуске редактор PyEdit будет импортировать начальные значения, определяющие шрифт, цвета, размеры текстового окна и необходимость учета регистра символов в операциях поиска. Настройки шрифта и цветов могут изменяться интерактивно с помощью меню, а окна позволяют свободно изменять их размер, поэтому данные настройки предусмотрены в значительной степени для удобства. Обратите также внимание, что этот модуль с настройками будет подключаться всеми экземплярами PyEdit, если он окажется доступным для импортирования клиентским программам, – даже при использовании редактора во всплывающих окнах или при встраивании в другие графические интерфейсы. Однако при необходимости клиентские приложения могут определять собственные версии этого файла с настройками или изменять существующий в пути поиска.

Изменения в версии PyEdit 2.1 (четвертое издание)

Помимо изменений, описанных в предыдущем разделе, при подготовке текущего четвертого издания в PyEdit были внесены следующие дополнительные улучшения:

- Редактор PyEdit был перенесен на новую версию Python 3.1 и его библиотеку tkinter.
- Были исправлены немодальные диалоги поиска с заменой и выбора шрифта – была обеспечена корректная их работа при наличии нескольких окон редактирования, за счет сохранения информации о каждом диалоге отдельно.
- При выполнении операции завершения, вызванной из главного окна, теперь проверяется наличие изменений в других окнах редактирования, открытых в пределах этого же процесса.
- Появился новый пункт Grep меню и диалог поиска во внешних файлах – поиск поддерживает текст Юникода и производится в отдельном потоке выполнения, чтобы избежать блокирования графического интерфейса и позволить одновременно выполнять несколько операций поиска.
- Было внесено небольшое исправление в начальное позиционирование, когда текст изначально вставляется во вновь созданное окно редактора, вызванное изменением в базовых библиотеках.
- Пункт Run Code меню для запуска файлов теперь использует не полное имя файла, содержащее полный путь к нему, а только базовую его часть, чтобы обеспечить поддержку относительных путей; позволяет определять аргументы командной строки для запускаемых файлов; и наследует исправление, выполненное в главе 5 в модуле launchmodes, которое преобразует / в \ в строках путей в файловой системе. Кроме того, этот пункт теперь всегда вызывает метод update между диалогами, чтобы гарантировать корректное отображение.
- Но самое заметное, пожалуй, изменение заключается в том, что теперь PyEdit позволяет отображать и редактировать содержимое файлов в *любых кодировках*, до той степени, до которой это позволяет библиотека Tk. В частности, имена кодировок учитываются при открытии и сохранении файлов, при отображении текста в графическом интерфейсе и когда выполняется поиск по файлам в каталогах.

В следующих разделах приводятся дополнительные примечания к перечисленным изменениям.

Исправление проблемы состояния модальных диалогов

Диалог *поиска с заменой* в предыдущей версии сохранял свои поля ввода в объекте текстового редактора, а это означает, что для всех открытых диалогов поиска с заменой использовались самые последние созданные экземпляры полей ввода. Это могло приводить к аварийному завер-

шению программы при попытке выполнить поиск с помощью диалога, открытого ранее, если к этому моменту был закрыт диалог, открытый позднее, так как при закрытии виджеты уничтожаются, – неожиданное поведение диалога, которое существовало, по крайней мере, со второго издания и которое я склонен был отнести к категории ошибок в операторах, но оказалось, что все дело в сохранении состояния! Тот же эффект наблюдался в диалоге *выбора шрифта* – самый последний экземпляр диалога затирал данные экземпляров, открытых перед ним, однако его обработчик исключений предотвращал аварийное завершение программы (он выводил окно с сообщением об ошибке). Чтобы исправить ошибки в диалогах поиска с заменой и выбора шрифта, теперь поля ввода в каждом диалоге передаются их обработчикам в виде аргументов. Вместо этого можно было бы разрешить создавать только по одному экземпляру этих диалогов, но это решение менее функционально.

Проверка наличия изменений в других окнах того же процесса при завершении

Кроме того, редактор PyEdit имел обыкновение игнорировать наличие изменений в других окнах редактирования при закрытии главного окна. В соответствии с реализацией, щелчок на кнопке Quit во всплывающем окне приводит к закрытию только этого окна, но операция закрытия главного окна вызывает метод `quit` из библиотеки tkinter, который завершает всю программу. В предыдущей версии при закрытии любого окна выполнялась проверка наличия изменений только в этом окне, а остальные окна игнорировались – закрытие главного окна могло привести к потере изменений в других окнах, закрываемых автоматически при завершении программы.

Чтобы исправить этот недостаток, текущая версия сохраняет список всех открытых в процессе окон редактирования – при закрытии главного окна теперь выполняется проверка наличия изменений во всех окнах и при необходимости предлагается подтвердить завершение программы. Это решение не устраняет все возможные проблемы (оно не устраняет проблему потери изменений, когда завершение приложения производится с помощью виджетов, находящихся за пределами PyEdit), но это существенное улучшение. Более полное решение может заключаться в переопределении или перехвате вызова метода `quit` библиотеки tkinter. Однако, чтобы не углубляться в детали, я отложу эту тему до более позднего обсуждения в этом же разделе (смотрите обсуждение реализации обработки события `<Destroy>` ниже); кроме того, смотрите соответствующие комментарии в конце файла с исходным программным кодом PyEdit, где даются примечания к реализации.

Новый диалог Grep: поиск в дереве файлов с поддержкой Юникода и многопоточной модели выполнения

Кроме всего прочего, в меню Search появился новый пункт Grep, реализующий поиск по внешним файлам. Этот инструмент выполняет ска-

нирование целого дерева каталогов в поисках файлов с именами, соответствующими шаблону и содержащими указанную строку. Результаты поиска отображаются в новом немодальном окне с прокручиваемым списком, где выводятся имена файлов, номера и содержимое строк с найденными совпадениями. Щелчок на элементе списка открывает соответствующий файл в новом немодальном окне редактирования PyEdit, при этом автоматически выполняется переход к строке с совпадением и ее выделение. В этой реализации повторно используется программный код, написанный нами ранее:

- Утилита `find`, написанная нами в главе 6, выполняющая обход дерева каталогов
- Реализация списка с прокруткой из главы 9 для отображения результатов поиска
- Конструктор рядов с полями для форм ввода, созданный в главе 10, для создания немодального диалога ввода
- Существующая реализация всплывающего режима использования PyEdit для отображения содержимого файлов по запросу
- Существующий обработчик события перехода в PyEdit для перемещения к строке с найденным совпадением

Поддержка многопоточной модели выполнения в операции поиска. Чтобы избежать блокирования графического интерфейса в процессе поиска, поиск производится в параллельных потоках выполнения. Это позволяет запускать сразу несколько операций поиска и выполнять их параллельно (что особенно полезно, когда поиск выполняется в больших деревьях каталогов, таких как стандартная библиотека Python или полное дерево его исходных текстов). При этом применяются такие приемы и механизмы, как стандартные потоки выполнения, очереди и цикл обработки событий от таймера `after`, с которыми мы познакомились в главе 10, – потоки-производители, не имеющие отношения к графическому интерфейсу, отыскивают совпадения и помещают их в очередь, которая проверяется главным потоком, управляющим графическим интерфейсом, в цикле обработки событий от таймера.

В данной реализации цикл обработки событий от таймера запускается только при выполнении операции поиска, и для каждого поиска используются отдельный поток, отдельный цикл обработки событий от таймера и отдельная очередь. В одном процессе одновременно может выполняться множество потоков и циклов обработки событий от таймера, связанных с поиском, а также могут существовать другие независимые потоки, очереди и циклы обработки событий от таймера. Например, прикрепляемый компонент PyEdit в программе PyMailGUI, представленной в главе 14, может выполнять собственные операции поиска, в то время как программа PyMailGUI выполняет собственные потоки и очереди, используемые для отправки или приема электронной почты. Каждый цикл обработки событий от таймера управляется независимо

от процессора событий tkinter. Из-за упрощенной архитектуры в этом примере не используется универсальная реализация `threadtools` очереди обработчиков из главы 10. Дополнительные примечания о реализации потоков выполнения в операции поиска ищите в исходных текстах, что приводятся далее, и сравните их с файлом *_unthreadedtextEditor.py* в дереве примеров, где содержится версия PyEdit без поддержки многопоточной модели выполнения.

Поддержка Юникода. Если внимательно изучить реализацию поиска по внешним файлам, можно заметить, что она позволяет определять кодировку для всего дерева и обрабатывает любые исключения, связанные с ошибками декодирования символов, возникающими как при обработке содержимого файлов, так и при обработке имен файлов во время обхода дерева. Как мы узнали в главах 4 и 6, при открытии текстовых файлов в Python 3.X должны декодироваться с применением кодировки, указанной явно или используемой на текущей платформе по умолчанию. Это представляет определенную проблему для операции поиска по файлам, так как деревья каталогов могут содержать файлы с символами в различных кодировках.

Фактически в Windows в одном дереве каталогов часто можно встретить файлы с содержимым в кодировках ASCII, UTF-8 и UTF-16 (варианты `ANSI`, `Utf-8` и `Unicode` выбора кодировки в Notepad) и даже в других кодировках, особенно в каталогах, где сохраняются файлы, загруженные из Интернета или полученные по электронной почте. Операция открытия таких файлов с применением кодировки UTF-8 в Python 3.X будет приводить к исключениям, а при открытии их в двоичном режиме программа будет получать закодированный текст, в котором едва ли возможно будет отыскать совпадение с искомой строкой. Технически, чтобы выполнить сравнение, необходимо выполнить декодирование байтов, прочитанных из файла, или закодировать искомую строку в байты. При этом совпадение может быть обнаружено только при использовании согласованных кодировок.

Чтобы обеспечить возможность поиска в деревьях каталогов со смешанными кодировками, диалог `Grep` открывает файлы в текстовом режиме и позволяет вводить имя кодировки, которая будет использоваться для декодирования содержимого всех файлов в просматриваемом дереве каталогов. Для удобства поле ввода с именем кодировки предварительно заполняется значением по умолчанию для текущей платформы, так как этого часто бывает вполне достаточно. Чтобы выполнить поиск в дереве каталогов с файлами разных типов, пользователи могут выполнить несколько операций поиска, указывая различные имена кодировок. При поиске могут также возникать ошибки декодирования имен файлов, но они практически никак не обрабатываются в текущей версии: предполагается, что имена файлов удовлетворяют соглашениям, принятым в файловой системе на данной платформе, в противном случае это приводит к завершению поиска (дополнительные сведения об утилите `find` обхода дерева каталогов, повторно используемой здесь, а также о проб-

лемах, связанных с кодированием имен файлов в Python, вы найдете в главах 4 и 6).

Кроме того, реализация операции Grep должна предусматривать обработку исключений, связанных с ошибками декодирования файлов, имена которых соответствуют шаблону, но содержимое не может быть декодировано с применением указанной кодировки, и фактически вообще может не быть текстом. Например, операция поиска в стандартной библиотеке Python 3.1 (как в примере поиска строки %, описанном выше) сталкивается с несколькими файлами, которые не смогли быть корректно декодированы в Windows на моем компьютере и могли бы вызвать крах PyEdit. Двоичные файлы, имена которых по случайности соответствуют шаблону, являются еще более худшим вариантом.

В целом программы могут избежать ошибок, вызванных применением неправильных кодировок, либо обрабатывая исключения, либо открывая файлы в двоичном режиме. Так как операция поиска может оказаться не в состоянии интерпретировать содержимое некоторых файлов как текст вообще, при ее реализации был выбран первый подход. В действительности, открытие даже текстовых файлов в двоичном режиме и чтение из них строк двоичных байтов в версии 3.X имитирует поведение текстовых файлов в версии 2.X и позволяет понять, почему принудительный переход на использование Юникода иногда является благом, – двоичный режим позволяет избежать появления исключений, связанных с декодированием, но сам текст по-прежнему остается закодированным и не может использоваться в привычных операциях. В этом случае операция сравнения может давать неверные результаты.

Дополнительные детали, касающиеся поддержки Юникода в реализации диалога Grep, а также описание проблем, связанных с этой поддержкой, и способов их решения, вы найдете в исходном программном коде, который приводится ниже. Дополнительные предложения по улучшению можно найти в главе 19, в описании модуля re – инструмента, который можно использовать для организации поиска по шаблону, а не только по определенной строке.

Исправление проблемы начального позиционирования

В этой версии текстовый редактор также обновляет свой графический интерфейс перед вставкой текста в текстовый виджет на этапе конструирования, когда ему передается имя файла в аргументе loadFirst. Спустя некоторое время, после выхода третьего издания и версии Python 2.5, в Tk или tkinter были внесены какие-то изменения, в результате такая операция вставки текста перед вызовом метода update стала приводить к прокручиванию виджета на одну строку – текст вставлялся, начиная со второй строки, а не с первой. Эта же проблема наблюдалась в версии для третьего издания, при использовании Python 2.6, но не 2.5. Добавление вызова метода update обеспечило корректное позиционирование

текстового виджета. Это неприятно, но такое вполне может происходить в мире, зависящем от внешних библиотек![1]

Клиенты, использующие классы редактора, также должны вызывать метод `update` перед вставкой текста вручную во вновь созданный (или скомпонованный) объект текстового редактора, чтобы обеспечить более точное позиционирование, – программа PyView, рассматриваемая далее в этой главе, и PyMailGUI в главе 14 учитывают эту особенность. Редактор PyEdit не может обновлять себя при каждом создании, потому что он может создаваться или даже скрываться вмещающими его графическими интерфейсами (например, это могло бы привести к отображению неполного окна в PyView). Кроме того, PyEdit мог бы автоматически обновлять себя в начале метода `setAllText`, чтобы исключить необходимость выполнять этот шаг клиентами, но принудительный вызов `update` требуется выполнить только один раз после компоновки (а не перед каждой вставкой текста), а кроме того, в некоторых случаях это могло бы приводить к нежелательным эффектам. Как правило, добавление лишних операций в методы, как в данном случае, обычно ограничивает область применения компонентов.

Улучшения в операции запуска программного кода

В реализацию операции `Run Code` из меню `Tools` было внесено три исправления, которые сделали еще более удобным запуск редактируемого программного кода из внешнего файла:

1. После перехода в каталог, где хранится файл, для обеспечения правильности всех относительных путей к файлам в его программном коде редактор PyEdit теперь отбрасывает путь из имени файла, прежде чем запустить его, потому что оригинальный путь к файлу может оказаться ошибочным, если он был относительным, а не абсолютным. Например, пути к файлам, открываемым вручную, являются абсолютными, но пути к файлам в программном коде PyDemos, вызывающем редактор, являются относительными – они откладываются относительно корневого каталога с примерами и после выполнения команды `chdir` могут оказаться недействительными.
2. Теперь в режиме запуска файла с программным кодом PyEdit использует инструменты запуска, поддерживающие возможность передачи аргументов командной строки в Windows.

[1] Интересно отметить, что даже текстовый редактор IDLE в Python 3.1 страдает от тех же двух ошибок, описываемых здесь и ликвидированных в текущей версии PyEdit, – IDLE в версии 3.1 вставляет содержимое файла при открытии начиная со второй строки, а его операция поиска во внешних файлах (напоминающая диалог Grep в PyEdit) вызывает крах в результате ошибки декодирования при просмотре стандартной библиотеки Python, что вызывает аварийное завершение IDLE. Здесь вполне уместно вспомнить поговорку про сапожника без сапог...

3. Редактор PyEdit унаследовал исправление, выполненное в модуле `launchmodes`, которое преобразует символы прямого слеша в обратные в строках путей в файловой системе (хотя позднее, из-за удаления префиксов относительных путей, полезность этого исправления стала вызывать сомнения). Необходимость преобразования прямых символов слеша в PyEdit обусловлена тем, что несмотря на допустимость их использования при вызове функции `open` в Windows, они не могут использоваться с некоторыми инструментами запуска в этой операционной системе.

Кроме того, в реализацию запуска программного кода как из внешних файлов, так и из строк в памяти, в этой версии был добавлен вызов метода `update` между вызовами диалогов, чтобы гарантировать, что последний диалог будет появляться на экране в любом случае (ранее в некоторых редких случаях второй диалог не отображался на экране). Даже с этими исправлениями операция `Run Code` по-прежнему не отличается надежностью. Например, при запуске программного кода из строки, а не из внешнего файла он выполняется внутри процесса, а не в отдельном потоке выполнения, и поэтому может заблокировать графический интерфейс. Кроме того, не совсем понятно, как лучше всего обрабатывать пути импортирования и каталоги для файлов при выполнении программного кода в виде строк, и стоит ли вообще сохранять этот режим. Измените эту особенность в соответствии со своими пожеланиями.

Поддержка текста Юникода (интернационализированного)

Наконец, из-за того, что теперь Python 3.X полностью поддерживает текст Юникода, редактор PyEdit также обеспечивает эту поддержку – он позволяет открывать, сохранять, просматривать, редактировать и отыскивать в дереве каталогов любой текст, в любой кодировке и с любыми наборами символов. Эта поддержка находит множество отражений в пользовательском интерфейсе PyMailGUI:

- При открытии файла у пользователя запрашивается имя кодировки (при этом предлагается системная кодировка по умолчанию), если она не была указана в настройках или при вызове редактора клиентским приложением.
- При сохранении нового файла запрашивается имя кодировки, если она не указана в настройках.
- При отображении и редактировании используется поддержка Юникода, реализованная в инструментах создания графических интерфейсов.
- Операция поиска в дереве каталогов позволяет явно указывать кодировку, которая будет применяться ко всем файлам в дереве, и пропускает файлы, декодировать которые не удалось, как было описано выше.

Благодаря этому обеспечивается поддержка интернационализированного текста, кодировка которого может отличаться от кодировки по умолчанию, используемой на текущей платформе. Это, в частности, удобно для просмотра текстовых файлов, полученных из Интернета, по электронной почте или через FTP. Приложение PyMailGUI из главы 14, например, использует встроенный объект PyEdit для отображения текста вложений различного происхождения и в различных кодировках. Поддержка Юникода в операции поиска по файлам была описана выше – остальные аспекты этой модели, по сути, сводятся к операциям открытия и сохранения файлов, как описывается в следующем разделе.

Файлы с текстом Юникода и модель отображения их содержимого. Поскольку строки в Python 3.X всегда интерпретируются как последовательности кодовых пунктов Юникода, поддержка Юникода в действительности означает поддержку различных кодировок при чтении и записи в текстовые файлы. Напомню, что текст может сохраняться в файлах в различных кодировках, – данные декодируются при чтении и кодируются при записи с применением этих кодировок. Если текст не всегда сохраняется в файлах с использованием кодировки по умолчанию для данной платформы, то для таких случаев нам необходимо знать, какую кодировку использовать при чтении и записи.

Чтобы обеспечить такую поддержку, редактор PyEdit использует подходы, подробно описанные в главе 9. Мы не будем повторно обсуждать их здесь, тем не менее в двух словах отмечу, что виджет `Text` принимает содержимое в виде строки типа `str` или `bytes` и всегда возвращает его как строку `str`. Редактор PyEdit отображает этот интерфейс на интерфейс объектов файлов языка Python следующим образом:

Входные файлы (открытие)

Чтобы декодировать байты из файла в строки, в общем случае требуется знать название кодировки, совместимой с данными в файле. Если кодировка окажется несовместимой, операция декодирования потерпит неудачу (например, при попытке декодировать 8-битовые данные с использованием кодировки ASCII). В некоторых случаях кодировка открываемого текстового файла может оказаться неизвестной.

Чтобы при загрузке содержимого входных файлов прочитать данные в виде строк `str`, редактор PyEdit сначала пытается открывать их в текстовом режиме, применяя кодировки, полученные из разных источников: из аргумента метода, когда кодировка известна заранее (например, из заголовков вложений в сообщениях электронной почты или из исходных файлов, открываемых демонстрационными примерами), из диалога, запрашивающего кодировку у пользователя, из модуля с настройками и из параметров по умолчанию текущей платформы. При выводе диалога, запрашивающего кодировку при открытии файла, поле ввода предварительно заполняется

вариантом из файла с настройками, который считается значением по умолчанию.

Если с помощью всех этих кодировок не удается декодировать файл, он открывается в двоичном режиме и текст из него читается как строка `bytes`, без декодирования, что фактически перемещает задачу декодирования в библиотеку Tk. В этом случае в Windows все последовательности `\r\n` вручную преобразуются в символы `\n`, чтобы обеспечить корректное отображение текста и последующее сохранение его в файл. Двоичный режим используется только в самом крайнем случае, чтобы лишний раз не полагаться на логику декодирования и ограниченную поддержку кодировок в библиотеке Tk.

Обработка текста

При обращении к виджету `Text` он возвращает свое содержимое в виде строки `str`, независимо от того, в каком виде, `str` или `bytes`, был вставлен текст. Вследствие этого вся обработка текстового содержимого производится с применением методов строк `str` Юникода.

Выходные файлы (сохранение)

Операция кодирования строк в байты при записи в файлы обычно отличается большей гибкостью, чем операция декодирования. При этом не требуется использовать ту же самую кодировку, которая применялась для декодирования данных в строку, но и эта операция может потерпеть неудачу, если выбранная схема кодирования окажется слишком узкой для содержимого строки (например, попытка кодировать 8-битовый текст с применением кодировки ASCII).

Для сохранения текста в файл редактор PyEdit открывает выходной файл в текстовом режиме, чтобы обеспечить отображение символов конца строки и кодирование содержимого строки `str` Юникода. Имя кодировки извлекается из одного из источников – это может быть кодировка, использовавшаяся при открытии или первоначальном сохранении файла (если была указана), или кодировка, полученная из диалога с пользователем, из модуля с настройками или из параметров по умолчанию текущей платформы. В отличие от операции открытия, когда операция сохранения выводит диалог запроса имени кодировки, поле ввода заполняется именем известной кодировки, если она была определена прежде, в противном случае берется вариант из файла с настройками, как и в случае с операцией открытия.

Диалоги ввода кодировки, вызываемые операциями открытия и сохранения файлов, – это лишь одно из воплощений описанных правил в графическом интерфейсе; другие варианты определяются в модуле с настройками. Поскольку заранее невозможно предугадать все возможные случаи использования, в редакторе PyEdit применяется либеральный подход: он поддерживает все мыслимые режимы и обеспечивает пользователям возможность влиять на определение кодировок с помощью определения настроек в их собственном модуле `textConfig`. Он пытается

применить одну кодировку за другой из разных источников, если это разрешено в модуле textConfig, пока не будет найдена кодировка, дающая положительные результаты. Тем самым достигается максимальная маневренность перед лицом переменчивого мира Юникода.

Например, согласно параметрам в файле с настройками операция сохранения повторно использует кодировку, которая применялась при открытии файла или при первой операции сохранения, если она известна. Для новых файлов (созданных выбором пункта New в меню или вставкой текста вручную) и для файлов, открытых в двоичном режиме, кодировка остается неизвестной до момента сохранения, но для файлов, которые удалось открыть в текстовом режиме, она известна. Кроме того, с помощью параметров в файле с настройками мы можем определить необходимость запрашивать кодировку у пользователя при выполнении операции Save As (и, возможно, Save), потому что у него могут быть свои предпочтения при создании новых файлов. Мы также можем запрашивать кодировку при открытии существующих файлов, потому что для этого необходимо знать его текущую кодировку. В некоторых случаях (например, для файлов, полученных из Интернета) пользователь может не знать ее, но в других случаях он может предпочесть указать кодировку явно. Вместо того чтобы выбирать тот или иной порядок действий в таких ситуациях, мы просто опираемся на пользовательские настройки.

На практике все это относится только к клиентам PyEdit, которые запрашивают начальную загрузку файлов или позволяют открывать и сохранять файлы с помощью графического интерфейса. Поскольку содержимое может вставляться как строка типа str или bytes, клиенты всегда имеют возможность читать входные файлы самостоятельно, до создания объекта текстового редактора, и вставлять в него текст вручную. Кроме того, клиенты могут получать содержимое вручную и сохранять его любым предпочитаемым способом. Такое выполнение операций вручную может оказаться полезным, если в каком-то контексте методика, реализованная в редакторе PyEdit, окажется нежелательной. Поскольку виджет Text всегда возвращает содержимое в виде строки str, остальной части этой программы безразлично, строка какого типа была в него вставлена.

Имейте в виду, что описанная методика по-прежнему зависит от поддержки Юникода и от ограничений, заложенных в библиотеке Tk, а также от интерфейса tkinter к ней. Редактор PyEdit позволяет загружать и сохранять текст в любой кодировке, но он не может гарантировать, что библиотека графического интерфейса сможет отобразить такой текст. То есть даже если мы совершенно корректно обрабатываем текст Юникода на стороне Python, мы все равно остаемся во власти других слоев программного обеспечения, обсуждение которых выходит далеко за рамки этой книги. Библиотека Tk достаточно надежно работает с самыми разными наборами символов, если ей передавать уже

декодированные строки str Юникода (смотрите, например, описание поддержки интернационализации в PyMailGUI в главе 14), но ситуации в конкретных случаях могут сильно различаться.

Порядок выбора кодировки и возможные варианты. Имейте также в виду, что политика редактора PyEdit в отношении Юникода отражает предпочтения единственного текущего пользователя и не проверялась на универсальность и эргономику – будучи книжным примером, редактор не использует встроенную среду тестирования, как это свойственно проектам с открытыми исходными текстами. Неплохие результаты можно было бы получать, используя другие схемы и порядки следования источников, и совершенно невозможно предугадать предпочтения каждого пользователя в каждом конкретном случае. Например:

- Непонятно, следует ли сначала запрашивать кодировку у пользователя, а потом пытаться использовать кодировку, указанную в файле с настройками, или наоборот.
- Возможно также, что мы всегда должны спрашивать кодировку у пользователя, полагаясь в основном на это, независимо от параметров настройки.
- При сохранении мы могли бы также попробовать самостоятельно определить кодировку для применения к строке str (например, попробовать применить UTF-8, Latin-1 или другую распространенную кодировку), но наши предположения могут не совпадать с тем, что имел в виду пользователь.
- Весьма вероятно, что пользователь пожелает сохранить файл в той же кодировке, которая применялась при открытии файла или при сохранении в первый раз. Редактор PyEdit предоставляет поддержку этого варианта, в противном случае графический интерфейс запрашивал бы кодировку для данного файла более чем один раз. Однако, поскольку некоторым пользователям может потребоваться повторно использовать операцию Save, чтобы сохранить тот же файл в другой кодировке, то предусмотрена возможность отключения этого варианта в модуле с настройками. На первый взгляд, для этой цели было бы лучше использовать операцию Save As, однако следующий пункт объясняет, почему это не всегда так.
- Точно так же неочевидно, должна ли операция Save As повторно использовать кодировку, которая применялась при открытии или при сохранении файла в первый раз, или она должна запрашивать новую кодировку – действительно ли при этом сохраняется совершенно новый файл или только копия предыдущего содержимого с установленной кодировкой, но под новым именем? Из-за такой неоднозначности мы даем возможность запретить использование установленной кодировки в операции Save As или в обеих операциях, Save и Save As, в модуле с настройками. По умолчанию использование известной кодировки разрешено только для операции Save и запрещено для Save As. В любом случае, операции сохранения выводят диалоги,

запрашивающие имя кодировки, в которых поле ввода заполнено известной кодировкой.

- Порядок выбора вариантов вообще выглядит весьма спорным. Например, возможно, операция Save As должна использовать известную кодировку, если настройки запрещают запрашивать ее у пользователя, – в данной реализации, если настройки запрещают использовать известную кодировку и запрашивать ее у пользователя, эта операция будет определять кодировку из файла с настройками или использовать системную кодировку по умолчанию (например, UTF-8), что может оказаться не самым лучшим решением при сохранении составных частей сообщений электронной почты, кодировка которых уже известна.

И так далее. Поскольку такой пользовательский интерфейс обеспечивает широчайший выбор вариантов, в целях иллюстрации в этой книге реализованы общий и частично эвристический алгоритмы поддержки каждого из возможных вариантов, и в качестве опоры при выборе используются настройки пользователя. Однако на практике такая гибкость может оказаться совершенно излишней – большинству пользователей наверняка будет достаточно поддержки какого-то одного алгоритма из числа поддерживаемых здесь.

Кроме того, вероятно, было бы удобнее, если бы алгоритмом выбора кодировки можно было управлять непосредственно в графическом интерфейсе, вместо того чтобы вручную определять его в модуле с настройками. Например, возможно, каждая операция – Open, Save и Save As – должна позволять выбирать кодировку и по умолчанию использовать последнюю известную кодировку, если таковая определена. Реализация этой возможности в виде раскрывающихся списков с именами кодировок или полей ввода в диалогах Save и Open позволила бы отказаться от лишних диалогов и достичь практически той же гибкости.

В текущей реализации редактора PyEdit имеется возможность определить в файле с настройками необходимость запрашивать кодировку у пользователя для обеих операций, открытия и сохранения, что дает практически тот же эффект, по крайней мере в тех ситуациях, с которыми я сталкивался до настоящего момента, и, возможно, является лучшим решением для большинства контекстов.

Таким образом, по умолчанию:

- Операция Open использует переданную ей кодировку, если она была указана, или запрашивает имя кодировки у пользователя
- Операция Save повторно использует известную кодировку, если она была определена ранее, а при сохранении новых файлов запрашивает ее у пользователя
- Операция Save As всегда запрашивает имя кодировки у пользователя, как при сохранении нового файла

- Операция Grep позволяет вводить кодировку в диалоге определения параметров поиска и применяет ее при поиске во всех файлах, имеющихся в дереве каталогов

С другой стороны, поскольку умолчаний, определяемых платформой, будет вполне достаточно для работы без лишних сложностей взаимодействия с графическим интерфейсом, по крайней мере, для подавляющего числа пользователей, с помощью параметров в модуле textConfig можно предотвратить вывод диалога запроса кодировки и вернуться к использованию кодировки, указанной явно или определяемой платформой по умолчанию. В конечном счете, определение наиболее удачного алгоритма выбора кодировки требует анализа предпочтений широкого круга пользователей, а не предположений единственного разработчика. Как всегда, вы свободно можете адаптировать этот алгоритм под свои потребности.

В подкаталоге *test* в дереве примеров вы найдете несколько текстовых файлов в различных кодировках, с которыми вы можете экспериментировать при изменении алгоритмов выбора кодировки в модуле textConfig для операций открытия и сохранения файлов. Этот каталог содержит файлы, изображенные на рис. 11.5 и 11.6, в которых используются национальные наборы символов и сохраненные в различных кодировках. Например, файл *email-part--koi8-r* содержит текст на русском языке, сохраненный в кодировке koi8-r, а файл *email-part--koi8-r--utf8* содержит тот же текст, сохраненный в кодировке UTF-8, – последний можно открыть в программе Блокнот (Notepad) в Windows, но первый будет корректно отображен только при передаче PyEdit явно указанной кодировки.

Еще лучше, сохраните сами один и тот же файл в нескольких кодировках, жестко определяя кодировку в модуле textConfig или указывая разные кодировки при сохранении, – благодаря широкомасштабной поддержке Юникода в Python 3.X, редактор PyEdit позволяет сохранять и загружать файлы практически в любой кодировке.

Еще о проверке наличия изменений при завершении: событие <Destroy>

Необходимо сказать несколько слов о еще одном изменении в версии 2.1, прежде чем перейти к программному коду, поскольку он иллюстрирует основы закрытия окон tkinter в действующей программе. В главе 8 мы узнали, что библиотека tkinter позволяет выполнять с помощью метода bind привязку обработчика к событию <Destroy>, которое возбуждается при закрытии окна или уничтожении виджета. Мы могли бы привязать обработчики этого события к окнам PyEdit или к их текстовым виджетам, чтобы перехватить момент завершения программы, но это не принесло бы нам никакой выгоды в данной ситуации. В обработчике этого события сценарии вообще не могут выполнять какие-либо опера-

ции с графическим интерфейсом, потому что к моменту его вызова графический интерфейс уже разрушен. В частности, попытка проверить наличие изменений в текстовом виджете или извлечь его содержимое в обработчике события <Destroy> может привести к исключению. Вывод диалога с сообщением о необходимости сохранения также может действовать несколько странно: он появится только после того, как некоторые виджеты окна уже будут стерты (включая текстовый виджет, содержимое которого пользователь должен был бы проверить и сохранить!), а иногда даже вообще может не появится.

Кроме того, как уже упоминалось в главе 8, вызов метода quit не возбуждает события <Destroy>, но вызывает фатальную ошибку Python при выходе. Чтобы вообще иметь возможность использовать события <Destroy>, при выполнении операции Quit редактор PyEdit должен был бы закрывать окна только вызовом метода destroy и полагаться на протокол закрытия корневого окна Tk – непосредственное завершение приложения оказалось бы невозможным или потребовало бы использования таких инструментов, как sys.exit. Поскольку в обработчике события <Destroy> любые операции с графическим интерфейсом оказываются недопустимыми, использование этого приема ничем не оправдано. Программный код, выполняемый после вызова функции mainloop, также не способен помочь решить эту проблему, потому что функция mainloop вызывается за пределами PyEdit и после выхода из нее оказывается слишком поздно проверять наличие изменений и выполнять сохранения.

Иными словами, событие <Destroy> не решает проблему проверки необходимости сохранения перед закрытием окна, и оно никак не помогает в случае вызова методов quit и destroy виджетов из-за пределов классов окон PyEdit. Из-за этих сложностей PyEdit полагается на проверку изменений перед закрытием в каждом отдельном окне и проверяет наличие изменений в окнах, находящихся в списке, прежде чем закрыть любое из главных окон. Приложения, следующие данной модели окон, будут выполнять проверку автоматически. Приложения, использующие PyEdit как компонент более крупного графического интерфейса или использующие его иными способами, управляя редактором PyEdit извне, сами должны проверять наличие несохраненных изменений при закрытии, еще до того, как объект PyEdit или его виджеты будут разрушены.

Чтобы поэкспериментировать с событием <Destroy>, отыщите в дереве примеров файл *destroyer.py* – он имитирует действия, которые должен был бы выполнить редактор PyEdit при получении события <Destroy>. Ниже приводится наиболее важный фрагмент из этого сценария с комментариями, поясняющими его поведение:

```
def onDeleteRequest():
    print('Got wm delete') # щелчок на кнопке X в окне: можно отменить
    root.destroy()         # возбудит событие <Destroy>
```

```
def doRootDestroy(event):
    print('Got event <destroy>') # для каждого виджета в корневом окне
    if event.widget == text:
        print('for text')
        print(text.edit_modified()) # <= ошибка Tcl: неверный виджет
        ans = askyesno('Save stuff?', 'Save?')    # <= некорректное поведение
        if ans: print(text.get('1.0', END+'-1c')) # <= ошибка Tcl: неверный
                                                  #                виджет

root = Tk()
text = Text(root, undo=1, autoseparators=1)
text.pack()
root.bind('<Destroy>', doRootDestroy)              # для корневого и дочерних
root.protocol('WM_DELETE_WINDOW', onDeleteRequest) # на кнопке X окна

Button(root, text='Destroy', command=root.destroy).pack() # возбудит <Destroy>
Button(root, text='Quit', command=root.quit).pack() # <= фатальная ошибка
mainloop()                                           # Python, quit() не
                                                     # возбуждает <Destroy>
```

Дополнительные подробности, касающиеся всего, о чем говорилось выше, ищите в листингах, которые приводятся в следующем разделе. Кроме того, обязательно прочитайте строку документирования в главном файле, где приводится список предлагаемых расширений и решений проблемы открытия файлов (под заголовком «TBD»). На реализацию редактора PyEdit значительное влияние оказали влияние мои личные предпочтения, но вы можете настроить его под себя.

Исходный программный код PyEdit

Программа PyEdit состоит из одного маленького модуля с настройками и одного главного файла с реализацией, содержащего чуть больше 1000 строк программного кода, с расширением *.py*, который можно запускать или импортировать. Для использования в Windows предоставляется еще один однострочный файл с расширением *.pyw*, который просто запускает файл *.py* вызовом `exec(open('textEditor.py').read())`. Расширение *.pyw* предотвращает появление консоли DOS на экране при запуске в Windows.

В настоящее время файлы с расширением *.pyw* могут импортироваться и выполняться как обычные файлы с расширением *.py* (их можно запускать двойным щелчком мыши или с помощью таких инструментов языка Python, как `os.system` и `os.startfile`), поэтому в действительности нет необходимости создавать отдельный файл, чтобы обеспечить возможность импортирования и запуска без вывода окна консоли. Однако я оставил расширение *.py*, чтобы в процессе разработки видеть сообщения, которые выводятся в окно консоли, и использовать PyEdit, как простую интегрированную среду разработки, – когда операция запуска программного кода настроена на выполнение отдельных инструкций

(а не файлов), вывод, который производится программным кодом, отображается в окне консоли DOS редактора PyEdit. Предполагается, что клиенты будут в обычном случае импортировать файл *.py*.

Файл с настройками пользователя

Итак, перейдем к программному коду. В первую очередь рассмотрим модуль с настройками пользователя, который приводится в примере 11.1. Он предназначен главным образом для того, чтобы было удобнее определять параметры внешнего вида, отличные от значений по умолчанию. Редактор PyEdit реализован так, что может работать и без этого модуля, и если он содержит синтаксические ошибки. Этот файл предназначен, прежде всего, для использования редактором PyEdit, когда он запускается как самостоятельный сценарий (в этом случае файл с настройками импортируется из текущего каталога), но вы также можете определить собственную версию файла с настройками PyEdit в любом другом каталоге, включенном в путь поиска модулей.

Дополнительно о том, какие настройки загружаются редактором, смотрите исходный программный код *textEditor.py* далее. Содержимое этого файла импортируется двумя различными способами – одна инструкция импортирования, которая загружает настройки внешнего вида, предполагает, что этот модуль (а не содержащий его пакет) находится в пути поиска модулей, и пропускает его, если его не находит, а другая, загружающая настройки порядка выбора кодировок, всегда отыскивает этот файл, независимо от способа запуска. Ниже описывается, что означает такое деление настроек для клиентов:

- Поскольку первая операция импортирования, загружающая настройки внешнего вида, ищет файл в пути поиска модулей, а не в каталоге основного пакета, то для каждого клиентского приложения, в его домашнем каталоге, можно определить собственный файл *textConfig.py* и тем самым обеспечить индивидуальные настройки PyEdit для каждого клиента.
- Настройки порядка выбора кодировки, напротив, всегда загружаются из файла, находящегося в каталоге пакета, с использованием операции импортирования по относительному пути, потому что они имеют более важное значение и маловероятно, что они будут отличаться от одного приложения к другому. Используемая здесь операция импортирования по относительному пути в пакете является эквивалентом импортирования всего пакета от корня *PP4E*, но не зависит от структуры каталогов.

Подобно эвристическим алгоритмам выбора кодировки символов, описанным выше, эта модель импортирования может считаться ориентировочной и может быть пересмотрена в соответствии с требованиями практической реализации.

Пример 11.1. PP4E\Gui\TextEditor\textConfig.py

```
"""
модуль с начальными настройками PyEdit (textEditor.py);
"""
#-----------------------------------------------------------------------------
# Общие настройки
# закомментируйте любые настройки в этом разделе, чтобы принять настройки по
# умолчанию библиотеки Tk или программы; шрифт/цвет можно также менять из меню
# в графическом интерфейсе, а также менять размеры окон после их открытия;
# импортируются из пути поиска модулей: могут определять отдельные настройки
# для каждого клиентского приложения, игнорируется, если находится не в пути
# поиска модулей;
#-----------------------------------------------------------------------------

# начальные настройки шрифта    # семейство, размер, стиль
font = ('courier', 9, 'normal') # например, стиль: 'bold italic'

# начальные настройки цвета     # по умолчанию = white, black
bg = 'lightcyan'                # название цвета или шестнадцатеричный код RGB
fg = 'black'                    # например, 'powder blue', '#690f96'

# начальные настройки размеров
height = 20                     # умолчания Tk: 24 строки
width = 80                      # умолчания Tk: 80 символов

# нечувствительность к регистру при поиске
caseinsens = True               # по умолчанию = 1/True (включена)

#-----------------------------------------------------------------------------
# 2.1: Порядок выбора кодировки для содержимого и имен файлов в операциях
# открытия и сохранения;
# опробует каждый случай из перечисленных ниже в указанном порядке, пока не
# будет обнаружен первый, дающий положительный результат; запишите во все
# переменные false/пустое значение/0, чтобы перейти к использованию умолчаний
# для вашей платформы (то есть 'utf-8' – в Windows, или 'ascii', 'latin-1'
# или другая кодировка в иных системах, таких как Unix);
# savesUseKnownEncoding: 0=Нет, 1=Да, только для операции Save, 2=Да для
# операций Save и SaveAs;
# всегда импортируются из этого файла: sys.path – если главный модуль, иначе –
# относительно пакета;
#-----------------------------------------------------------------------------

                    # 1) Сначала выполняется попытка применить известную
                    #    кодировку (например, из заголовка сообщения
                    #    электронной почты)
opensAskUser = True # 2) Если True, далее выполняется запрос у пользователя
                    #    (предварительно заполняется значением по умолчанию)
opensEncoding = ''  # 3) Если непустое значение, далее будет выполнена попытка
                    #    применить эту кодировку: 'latin-1', 'cp500'
```

```
                    # 4) Далее выполняется попытка применить
                    #    sys.getdefaultencoding() - системное значение
                    #    по умолчанию
                    # 5) В крайнем случае текст передается в двоичном виде и
                    #    используются алгоритмы Tk

savesUseKnownEncoding = 1 # 1) Если > 0, выполняется попытка применить
                          #    кодировку, известную по последней операции Open
                          #    или Save
savesAskUser = True       # 2) Если True, далее выполняется запрос у
                          #    пользователя (предварительно заполняется
                          #    известным значением?)
savesEncoding = ''        # 3) Если непустое значение, далее будет выполнена
                          #    попытка применить эту кодировку: 'utf-8' и так
                          #    далее
                          # 4) В крайнем случае выполняется попытка применить
                          #    sys.getdefaultencoding()
```

Файлы запуска для Windows (и других систем)

Далее, в примере 11.2 приводится файл запуска с расширением *.pyw*, используемый для подавления окна консоли DOS в Windows, которое выводится при запуске в некоторых режимах (например, двойным щелчком). Окно консоли по-прежнему можно получить при запуске файла с расширением *.py* (например, чтобы увидеть вывод, генерируемый программным кодом, запускаемым редактором в режиме выполнения единственной инструкции). Двойной щелчок на этом файле дает тот же эффект, что и запуск PyEdit с помощью панели запуска PyDemos или PyGadgets.

Пример 11.2. PP4E\Gui\TextEditor\textEditorNoConsole.pyw

```
"""
запускает редактор и подавляет вывод окна консоли DOS в Windows; с тем же
успехом можно было бы просто присвоить расширение .pyw основному файлу, что не
помешало бы возможности импортировать его, но файл .py был оставлен, чтобы иметь
возможность наблюдать вывод в консоли
"""

exec(open('textEditor.py').read()) # как будто содержимое файла вставляется
                                   # сюда (или textEditor.main())
```

Пример 11.2 прекрасно справляется с возложенной на него задачей, но когда я обновлял эту книгу, мне надоело использовать каждый раз Notepad для просмотра текстовых файлов из командных строк, запускаемых из произвольных мест, поэтому я написал сценарий, представленный в примере 11.3, который запускает PyEdit более универсальным и автоматизированным способом. Этот сценарий подавляет вывод консоли DOS, подобно примеру 11.2, когда запускается щелчком мыши на ярлыке в Windows, но дополнительно выполняет настройку пути

поиска модулей на компьютерах, где я не использовал Панель Управления (Control Panel) для этого, и позволяет запускать редактор, даже когда он находится за пределами текущего рабочего каталога.

Пример 11.3. PP4E\Gui\TextEditor\pyedit.pyw

```
#!/usr/bin/python
"""
удобный сценарий для запуска pyedit из произвольного каталога, выполняет
необходимую корректировку пути поиска модулей; sys.path при импортировании
и функции open() требуется передавать путь относительно известного пути
к каталогу со сценарием, а не относительно текущего рабочего каталога, потому
что текущим является каталог сценария, только если сценарий запускается щелчком
на ярлыке, а при вводе команды в командной строке он может находиться в любом
другом каталоге: использует путь из argv; этому файлу дано расширение .pyw,
чтобы подавить вывод окна консоли в Windows; добавьте каталог с этим сценарием
в системную переменную PATH, чтобы иметь возможность запускать его из командной
строки; также может использоваться в Unix: символы / и \ обрабатываются
переносимым образом;
"""

import sys, os
mydir = os.path.dirname(sys.argv[0])  # использовать каталог сценария для
                                      # open, sys.path
sys.path.insert(1, os.sep.join([mydir] + ['..']*3)) # импорт: PP4E - корень,
                                                    # 3 уровнями выше
exec(open(os.path.join(mydir, 'textEditor.py')).read())
```

Чтобы запустить его из командной строки в окне консоли, достаточно, чтобы путь к каталогу со сценарием находился в системной переменной окружения PATH, – действие, выполняемое в первой строке в следующем фрагменте, достаточно было бы выполнить один раз в Панели Управления (Control Panel) Windows:

```
C:\...\PP4E\Internet\Web> set PATH=%PATH%;C:\...\PP4E\Gui\TextEditor
C:\...\PP4E\Internet\Web> pyedit.pyw test-cookies.py
```

Этот сценарий также работает и в Unix, хотя в нем нет необходимости, если правильно установить переменные окружения PYTHONPATH и PATH (после этого можно запускать *textEditor.py* непосредственно), – а я не стал выполнять эти настройки на всех моих компьютерах. Ради интереса можно попробовать зарегистрировать этот сценарий как средство автоматического открытия файлов «.txt» при щелчке на них или при вводе их имен в командной строке (если, конечно, вы спокойно перенесете расставание с Notepad).

Реализация главного файла

Наконец, модуль в примере 11.4 представляет собой реализацию PyEdit. Этот файл может запускаться как самостоятельный сцена-

рий или импортироваться другими приложениями. Его программный код организован по пунктам главного меню. Главные классы, используемые для запуска и встраивания объекта PyEdit, находятся в конце файла. Во время экспериментов с PyEdit изучайте этот листинг, чтобы разобраться в его возможностях и используемых приемах.

Пример 11.4. PP4E\Gui\TextEditor\textEditor.py

```
"""
################################################################################
PyEdit 2.1: Текстовый редактор и компонент на Python/tkinter.

Использует текстовый виджет из библиотеки Tk, меню и панель инструментов
GuiMaker для реализации полнофункционального текстового редактора, который
может выполняться, как самостоятельная программа, или прикрепляться к другим
графическим интерфейсам, как компонент. Используется также в PyMailGUI
и PyView для редактирования сообщений электронной почты и примечаний к файлам
изображений. Кроме того, используется в PyMailGUI и PyDemos во всплывающем
режиме для отображения текстовых файлов и файлов с исходными текстами.

Новое в версии 2.1 (4 издание)
- работает под управлением Python 3.X (3.1)
- добавлен пункт "grep" меню и диалог: многопоточный поиск в файлах
- проверяет все окна на наличие несохраненных изменений при завершении
- поддерживает произвольные кодировки для файлов: в соответствии с настройками
  в файле textConfig.py
- переработаны диалоги поиска с заменой и выбора шрифта, чтобы обеспечить
  возможность одновременного вывода нескольких диалогов
- вызывает self.update() перед вставкой текста в новое окно
- различные улучшения в реализации операции Run Code, как описывается
  в следующем разделе

2.1 улучшения в реализации операции Run Code:
- после команды chdir использует базовое имя запускаемого файла, а не
  относительные пути
- в Windows использует инструмент запуска, поддерживающий передачу аргументов
  командной строки
- операция Run Code наследует преобразование символов обратного слеша от
  модуля launchmodes (необходимость в этом уже отпала)

Новое в версии 2.0 (3 издание)
- добавлен простой диалог выбора шрифта
- использует прикладной интерфейс Tk 8.4 к стеку отмен, чтобы добавить
  поддержку отмены/возврата (undo/redo) операций редактирования
- запрос подтверждения при выполнении операций Quit, Open, New, Run
  выполняется, только если имеются несохраненные изменения
- поиск теперь по умолчанию выполняется без учета регистра символов
- создан модуль с настройками для начальных значений
  шрифта/цвета/размера/чувствительности к регистру при поиске
```

```
TBD[1] (и предложения для самостоятельной реализации):
- необходимость учета регистра символов при поиске можно было бы задавать в
  графическом интерфейсе (а не только в файле с настройками)
- при поиске по файлу или в операции Grep можно было бы использовать поддержку
  регулярных выражений, реализованную в модуле re  (см. следующую главу)
- можно было бы попробовать реализовать подсветку синтаксиса (как в IDLE или в
  других редакторах)
- можно было бы попробовать проверить завершение работы программы методом
  quit() в неподконтрольных окнах
- можно было бы помещать в очередь каждый результат, найденный в диалоге Grep,
  чтобы избежать задержек
- можно было бы использовать изображения на кнопках в панели инструментов (как
  в примерах из главы 9)
- можно было бы просматривать строки, чтобы определить позицию вставки Tk для
  оформления отступов в окне Info
- можно было бы поэкспериментировать с проблемой кодировок в диалоге "grep"
  (смотрите примечания в программном коде);
################################################################################
"""

Version = '2.1'
import sys, os                                  # платформа, аргументы,
                                                # инструменты запуска
from tkinter import *                           # базовые виджеты, константы
from tkinter.filedialog import Open, SaveAs     # стандартные диалоги
from tkinter.messagebox import showinfo, showerror, askyesno
from tkinter.simpledialog import askstring, askinteger
from tkinter.colorchooser import askcolor
from PP4E.Gui.Tools.guimaker import *           # Frame + построители
                                                # меню/панелей инструментов
# общие настройки
try:
    import textConfig             # начальный шрифт и цвета
    configs = textConfig.__dict__ # сработает, даже если модуль отсутствует в
except:                           # пути поиска или содержит ошибки
    configs = {}

helptext = """PyEdit, версия %s
апрель, 2010
(2.0: январь, 2006)
(1.0: октябрь, 2000)

Программирование на Python, 4 издание
Марк Лутц (Mark Lutz), для издательства O'Reilly Media, Inc.

Программа и встраиваемый компонент текстового редактора,
написанный на Python/tkinter. Для быстрого доступа к операциям
использует отрывные меню, панели инструментов и горячие клавиши в меню.
```

[1] To Be Done – что еще можно сделать. – *Прим. ред.*

```
Дополнения в версии %s:
- поддержка 3.X
- новый диалог "grep" поиска во внешних файлах
- проверка несохраненных изменений при завершении
- поддержка произвольных кодировок для файлов
- допускает одновременный вывод нескольких диалогов
  поиска с заменой и выбора шрифта
- различные улучшения в операции Run Code

Дополнения в предыдущей версии:
- диалог выбора шрифта
- неограниченное количество отмен/возвратов
- quit/open/new/run предлагают сохранить, только
  если есть несохраненные изменения
- поиск выполняется без учета регистра символов
- модуль с начальными настройками textConfig.py
"""

START     = '1.0'             # индекс первого символа: строка=1,столбец=0
SEL_FIRST = SEL + '.first'    # отобразить тег sel в индекс
SEL_LAST  = SEL + '.last'     # то же, что 'sel.last'

FontScale = 0                 # использовать увеличенный шрифт в Linux
if sys.platform[:3] != 'win': # и в других не-Windows системах
    FontScale = 3
################################################################################
# Главные классы: реализуют графический интерфейс редактора, операции
# разновидности GuiMaker должны подмешиваться в более специализированные
# подклассы, а не наследоваться непосредственно, потому что этот класс
# принимает множество форм.
################################################################################

class TextEditor:  # смешать с классом Frame, имеющим меню/панель инструментов
    startfiledir = '.' # для диалогов
    editwindows = []   # для проверки при завершении

    # Настройки порядка выбора кодировки
    # импортируется в класс, чтобы обеспечить возможность переопределения в
    # подклассе

    if __name__ == '__main__':
        from textConfig import (   # мой каталог в пути поиска
            opensAskUser, opensEncoding,
            savesUseKnownEncoding, savesAskUser, savesEncoding)
    else:
        from .textConfig import (  # 2.1: всегда из этого пакета
            opensAskUser, opensEncoding,
            savesUseKnownEncoding, savesAskUser, savesEncoding)

    ftypes = [('All files',   '*'),     # для диалога открытия файла
              ('Text files',   '.txt'), # настроить в подклассе или
```

```
          ('Python files', '.py')]  # устанавливать в каждом экземпляре

colors = [{'fg':'black',      'bg':'white'}, # список цветов для выбора
          {'fg':'yellow',     'bg':'black'}, # первый элемент по умолчанию
          {'fg':'white',      'bg':'blue'},  # переделать по-своему или
          {'fg':'black',      'bg':'beige'}, # использовать элемент выбора
          {'fg':'yellow',     'bg':'purple'},# PickBg/Fg
          {'fg':'black',      'bg':'brown'},
          {'fg':'lightgreen', 'bg':'darkgreen'},
          {'fg':'darkblue',   'bg':'orange'},
          {'fg':'orange',     'bg':'darkblue'}]

fonts = [('courier',    9+FontScale, 'normal'), # шрифты, нейтральные
         ('courier',   12+FontScale, 'normal'), # в отношении платформы
         ('courier',   10+FontScale, 'bold'),   # (семейство, размер, стиль)
         ('courier',   10+FontScale, 'italic'), # или вывести в списке
         ('times',     10+FontScale, 'normal'), # увеличить в Linux
         ('helvetica', 10+FontScale, 'normal'), # использовать
         ('ariel',     10+FontScale, 'normal'), # 'bold italic' для 2
         ('system',    10+FontScale, 'normal'), # а также 'underline'
         ('courier',   20+FontScale, 'normal')]

def __init__(self, loadFirst='', loadEncode=''):
    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.knownEncoding = None # 2.1 кодировки: заполняется Open или Save
    self.text.focus()         # иначе придется щелкнуть лишний раз
    if loadFirst:
        self.update()                      # 2.1: иначе строка 2;
        self.onOpen(loadFirst, loadEncode) # см. описание в книге

def start(self):                        # вызывается из GuiMaker.__init__
    self.menuBar = [                    # настройка меню/панелей
        ('File', 0,                     # определение дерева меню GuiMaker
            [('Open...',    0, self.onOpen), # встроен. метод для self
             ('Save',       0, self.onSave), # метка, клавиша, обработчик
             ('Save As...', 5, self.onSaveAs),
             ('New',        0, self.onNew),
             'separator',
             ('Quit...',    0, self.onQuit)]
        ),
        ('Edit', 0,
            [('Undo',       0, self.onUndo),
             ('Redo',       0, self.onRedo),
             'separator',
             ('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),
                 ('Grep...',   3, self.onGrep)]
            ),
            ('Tools', 0,
                [('Pick Font...', 6, self.onPickFont),
                 ('Font List',    0, self.onFontList),
                 'separator',
                 ('Pick Bg...',   3, 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):                    # вызывается из GuiMaker.__init__
        name = Label(self, bg='black', fg='white') # ниже меню, выше панели
        name.pack(side=TOP, fill=X)                # компоновка меню/панелей
                                                   # фрейм GuiMaker
                                                   # компонуется сам
        vbar = Scrollbar(self)
        hbar = Scrollbar(self, orient='horizontal')
        text = Text(self, padx=5, wrap='none') # запретить перенос строк
        text.config(undo=1, autoseparators=1)  # 2.0, по умолчанию 0, 1

        vbar.pack(side=RIGHT,  fill=Y)
        hbar.pack(side=BOTTOM, fill=X)         # скомпоновать Text последним
        text.pack(side=TOP,    fill=BOTH, expand=YES) # иначе обрежутся полосы
                                                      # прокрутки
        text.config(yscrollcommand=vbar.set)   # вызывать vbar.set при
        text.config(xscrollcommand=hbar.set)   # перемещении по тексту
        vbar.config(command=text.yview) # вызывать text.yview при прокрутке
        hbar.config(command=text.xview) # или hbar['command']=text.xview
```

```
    # 2.0: применить пользовательские настройки или умолчания
    startfont = configs.get('font', self.fonts[0])
    startbg   = configs.get('bg',   self.colors[0]['bg'])
    startfg   = configs.get('fg',   self.colors[0]['fg'])
    text.config(font=startfont, bg=startbg, fg=startfg)
    if 'height' in configs: text.config(height=configs['height'])
    if 'width'  in configs: text.config(width =configs['width'])
    self.text = text
    self.filelabel = name

##############################################################################
# Операции меню File
##############################################################################

def my_askopenfilename(self): # объекты запоминают каталог/файл
    if not self.openDialog:   # последней операции
        self.openDialog = Open(initialdir=self.startfiledir,
                               filetypes=self.ftypes)
    return self.openDialog.show()

def my_asksaveasfilename(self): # объекты запоминают каталог/файл
    if not self.saveDialog:     # последней операции
        self.saveDialog = SaveAs(initialdir=self.startfiledir,
                                 filetypes=self.ftypes)
    return self.saveDialog.show()

def onOpen(self, loadFirst='', loadEncode=''):
    """
    2.1: полностью переписан для поддержки Юникода; открывает в текстовом
    режиме с кодировкой, переданной в аргументе, введенной пользователем,
    заданной в модуле textconfig или с кодировкой по умолчанию;
    в крайнем случае открывает файл в двоичном режиме и отбрасывает
    символы \r в Windows, если они присутствуют, чтобы обеспечить
    нормальное отображение текста; содержимое извлекается и возвращается в
    виде строки str, поэтому при сохранении его требуется кодировать:
    сохраняет кодировку, используемую здесь;

    предварительно проверяет возможность открытия файла;
    мы могли бы также вручную загружать и декодировать bytes в str, чтобы
    избежать необходимости выполнять несколько попыток открытия, но этот
    прием подходит не для всех случаев;

    порядок выбора кодировки настраивается в локальном textConfig.py:
    1) сначала применяется кодировка, переданная клиентом (например,
       кодировка из заголовка сообщения электронной почты)
    2) затем, если opensAskUser возвращает True, применяется кодировка,
       введенная пользователем (предварительно в диалог записывается
       кодировка по умолчанию)
    3) затем, если opensEncoding содержит непустую строку, применяется
       эта кодировка: 'latin-1' и так далее.
```

```
4) затем  выполняется попытка применить кодировку
   sys.getdefaultencoding()
5) в крайнем случае выполняется чтение в двоичном режиме и
   используется алгоритм, заложенный в библиотеку Tk
"""

if self.text_edit_modified(): # 2.0
    if not askyesno('PyEdit', 'Text has changed: discard changes?'):
        return

file = loadFirst or self.my_askopenfilename()
if not file:
    return

if not os.path.isfile(file):
    showerror('PyEdit', 'Could not open file ' + file)
    return

# применить известную кодировку, если указана
# (например, из заголовка сообщения электронной почты)
text = None     # пустой файл = '' = False: проверка на None!
if loadEncode:
    try:
        text = open(file, 'r', encoding=loadEncode).read()
        self.knownEncoding = loadEncode
    except (UnicodeError, LookupError, IOError): # Lookup: ошибка
        pass                                     # в имени

# применить кодировку, введенную пользователем,
# предварительно записать в диалог следующий вариант, как значение
# по умолчанию
if text == None and self.opensAskUser:
    self.update() # иначе в некоторых случаях диалог не появится
    askuser = askstring('PyEdit', 'Enter Unicode encoding for open',
                        initialvalue=(self.opensEncoding or
                                      sys.getdefaultencoding() or ''))
    if askuser:
        try:
            text = open(file, 'r', encoding=askuser).read()
            self.knownEncoding = askuser
        except (UnicodeError, LookupError, IOError):
            pass

# применить кодировку из файла с настройками (может быть, выполнять
# эту попытку до того, как запрашивать кодировку у пользователя?)
if text == None and self.opensEncoding:
    try:
        text = open(file, 'r', encoding=self.opensEncoding).read()
        self.knownEncoding = self.opensEncoding
    except (UnicodeError, LookupError, IOError):
        pass
```

```
        # применить системную кодировку по умолчанию (utf-8 в windows;
        # всегда пытаться использовать utf8?)
        if text == None:
            try:
                text = open(file, 'r',
                            encoding=sys.getdefaultencoding()).read()
                self.knownEncoding = sys.getdefaultencoding()
            except (UnicodeError, LookupError, IOError):
                pass

        # крайний случай: использовать двоичный режим и положиться на
        # возможности Tk
        if text == None:
            try:
                text = open(file, 'rb').read()      # строка bytes
                text = text.replace(b'\r\n', b'\n') # для отображения
                self.knownEncoding = None           # и последующего сохранения
            except IOError:
                pass

        if text == None:
            showerror('PyEdit', 'Could not decode and open file ' + file)
        else:
            self.setAllText(text)
            self.setFileName(file)
            self.text.edit_reset()     # 2.0: очистка стеков undo/redo
            self.text.edit_modified(0) # 2.0: сбросить флаг наличия изменений

    def onSave(self):
        self.onSaveAs(self.currfile) # may be None

    def onSaveAs(self, forcefile=None):
        """
        2.1: полностью переписан для поддержки Юникода: виджет Text всегда
        возвращает содержимое в виде строки str, поэтому нам необходимо
        побеспокоиться о кодировке, чтобы сохранить файл, независимо от
        режима, в котором открывается выходной файл (для двоичного режима
        необходимо будет получить bytes, а для текстового необходимо указать
        кодировку); пытается применить кодировку, использовавшуюся при
        открытии или сохранении (если известна), предлагаемую пользователем,
        указанную в файле с настройками, и системную кодировку по умолчанию;
        в большинстве случаев можно использовать системную кодировку по
        умолчанию;

        в случае успешного выполнения операции сохраняет кодировку для
        использования в дальнейшем, потому что это может быть первая операция
        Save после операции New или вставки текста вручную; в файле
        с настройками можно определить, чтобы  обе операции, Save и Save As,
        использовали последнюю известную кодировку (однако если для операции
        Save это оправданно, то в случае с операцией Save As это не так
        очевидно); графический интерфейс предварительно записывает эту
        кодировку в диалог, если она известна;
```

```
выполняет text.encode() вручную, чтобы избежать создания файла; для
текстовых файлов автоматически выполняется преобразование символов
конца строки: в Windows добавляются символы \r, отброшенные при
открытии файла в текстовом (автоматически) или в двоичном (вручную)
режиме; Если содержимое вставлялось вручную, здесь необходимо
предварительно удалить символы \r, иначе они будут продублированы;
knownEncoding=None перед первой операцией Open или Save, после New
и если операция Open открыла файл в двоичном режиме;

порядок выбора кодировки настраивается в локальном textConfig.py:
1) если savesUseKnownEncoding > 0, применить кодировку, использованную
   в последней операции Open или Save
2) если savesAskUser = True, применить кодировку, указанную
   пользователем (предлагать известную в качестве значения по
   умолчанию?)
3) если savesEncoding - непустая строка, применить эту кодировку:
   'utf-8' и так далее
4) в крайнем случае применить sys.getdefaultencoding()
"""
filename = forcefile or self.my_asksaveasfilename()
if not filename:
    return

text = self.getAllText() # 2.1: строка str, без символов \r,
encpick = None           # даже если текст читался/вставлялся
                         # в двоичном виде

# применить известную кодировку, использовавшуюся в последней операции
# Open или Save, если известна
if self.knownEncoding and (                              # известна?
   (forcefile and self.savesUseKnownEncoding >= 1) or   # для Save?
   (not forcefile and self.savesUseKnownEncoding >= 2)):# для SaveAs?
    try:
        text.encode(self.knownEncoding)
        encpick = self.knownEncoding
    except UnicodeError:
        pass

# применить кодировку, введенную пользователем,
# предварительно записать в диалог следующий вариант, как значение
# по умолчанию
if not encpick and self.savesAskUser:
    self.update()# иначе в некоторых случаях диалог не появится
    askuser = askstring('PyEdit', 'Enter Unicode encoding for save',
                        initialvalue=(self.knownEncoding or
                                      self.savesEncoding or
                                      sys.getdefaultencoding() or ''))
if askuser:
    try:
```

```
                text.encode(askuser)
                encpick = askuser
            except (UnicodeError, LookupError): # LookupError: ошибка в имени
                pass                            # UnicodeError: ошибка
                                                # кодирования
        # применить кодировку из файла с настройками
        if not encpick and self.savesEncoding:
            try:
                text.encode(self.savesEncoding)
                encpick = self.savesEncoding
            except (UnicodeError, LookupError):
                pass

        # применить системную кодировку по умолчанию (utf8 в windows)
        if not encpick:
            try:
                text.encode(sys.getdefaultencoding())
                encpick = sys.getdefaultencoding()
            except (UnicodeError, LookupError):
                pass

        # открыть в текстовом режиме, чтобы автоматически выполнить
        # преобразование символов конца строки и применить кодировку
        if not encpick:
            showerror('PyEdit', 'Could not encode for file ' + filename)
        else:
            try:
                file = open(filename, 'w', encoding=encpick)
                file.write(text)
                file.close()
            except:
                showerror('PyEdit', 'Could not write file ' + filename)
            else:
                self.setFileName(filename)   # может быть вновь созданным
                self.text.edit_modified(0)   # 2.0: сбросить флаг изменений
                self.knownEncoding = encpick # 2.1: запомнить кодировку
                                             # не сбрасывать стеки undo/redo!
    def onNew(self):
        """
        запускает редактирование совершенно нового файла в текущем окне;
        смотрите метод onClone, который вместо этого создает независимое окно
        редактирования;
        """
        if self.text_edit_modified():  # 2.0
            if not askyesno('PyEdit', 'Text has changed: discard changes?'):
                return
        self.setFileName(None)
        self.clearAllText()
        self.text.edit_reset()     # 2.0: очистить стеки undo/redo
        self.text.edit_modified(0) # 2.0: сбросить флаг наличия изменений
        self.knownEncoding = None  # 2.1: кодировка неизвестна
```

```
def onQuit(self):
    """
    вызывается выбором операции Quit в меню/панели инструментов и щелчком
    на кнопке X в заголовке окна;
    2.1: не завершать приложение при наличии несохраненных изменений;
    2.0: не выводить запрос на подтверждение, если нет изменений в self;
    перемещен в классы окон верхнего уровня ниже, так как его
    реализация может зависеть от особенностей использования: операция Quit
    в графическом интерфейсе может вызывать метод quit() для завершения,
    destroy() - чтобы просто закрыть окно Toplevel, Tk или фрейм
    с редактором, эта операция может даже вообще не предоставляться,
    если редактор присоединяется, как компонент; проверяет self на наличие
    несохраненных изменений, а если предполагается вызов метода quit(),
    главные окна должны также проверить наличие несохраненных изменений
    в других окнах, присутствующих в глобальном списке процесса;
    """
    assert False, 'onQuit must be defined in window-specific sublass'

def text_edit_modified(self):
    """
    2.1: теперь действует! кажется, проблема заключалась в типе bool
         результата в tkinter;
    2.0: self.text.edit_modified() не работает в Python 2.4: выполнить
         проверку вручную;
    """
    return self.text.edit_modified()
   #return self.tk.call((self.text._w, 'edit') + ('modified', None))

##############################################################################
# Операции меню Edit
##############################################################################

def onUndo(self):              # 2.0
    try:                       # tk8.4 поддерживает стеки undo/redo
        self.text.edit_undo() # возбуждает исключение, если стеки пустые
    except TclError:           # меню открывается для быстрого доступа
        showinfo('PyEdit', 'Nothing to undo') # к операциям

def onRedo(self):              # 2.0: возврат отмененной операции
    try:                       # редактирования
        self.text.edit_redo()
    except TclError:
        showinfo('PyEdit', 'Nothing to redo')

def onCopy(self):                        # получить текст, выделенный мышью
    if not self.text.tag_ranges(SEL):  # сохранить в системном 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) # вставить в текущую позицию 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 вырезать

def onSelectAll(self):
    self.text.tag_add(SEL, '1.0', END+'-1c')# выделить весь текст
    self.text.mark_set(INSERT, '1.0')       # переместить позицию в начало
    self.text.see(INSERT)                   # прокрутить в начало

########################################################################
# Операции меню Search
########################################################################

def onGoto(self, forceline=None):
    line = forceline or askinteger('PyEdit', 'Enter line number')
    self.text.update()
    self.text.focus()
    if line is not None:
        maxindex = self.text.index(END+'-1c')
        maxline  = int(maxindex.split('.')[0])
        if line > 0 and line <= maxline:
            self.text.mark_set(INSERT, '%d.0' % line)     # перейти к стр.
            self.text.tag_remove(SEL, '1.0', END)         # снять выделен.
            self.text.tag_add(SEL, INSERT, 'insert + 1l') # выделить стр.
            self.text.see(INSERT)                         # прокрутить
        else:                                             # до строки
            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:                                 # 2.0: без учета регистра символов
            nocase = configs.get('caseinsens', True)   # 2.0: настройки
            where = self.text.search(key, INSERT, END, nocase=nocase)
            if not where:                               # не переходить
                showerror('PyEdit', 'String not found')# в начало
            else:
                pastkey = where + '+%dc' % len(key)    # позиция после ключа
                self.text.tag_remove(SEL, '1.0', END)  # снять выделение
                self.text.tag_add(SEL, where, pastkey) # выделить ключ
                self.text.mark_set(INSERT, pastkey)    # для след. поиска
                self.text.see(where)                   # прокрутить экран

    def onRefind(self):
        self.onFind(self.lastfind)

    def onChange(self):
        """
        немодальный диалог поиска с заменой
        2.1: поля ввода диалога передаются обработчику, допускается открывать
        одновременно несколько диалогов поиска с заменой
        """
        new = Toplevel(self)
        new.title('PyEdit - change')
        Label(new, text='Find text?', relief=RIDGE, width=15).grid(row=0,
                                                                   column=0)
        Label(new, text='Change to?', relief=RIDGE, width=15).grid(row=1,
                                                                   column=0)
        entry1 = Entry(new)
        entry2 = Entry(new)
        entry1.grid(row=0, column=1, sticky=EW)
        entry2.grid(row=1, column=1, sticky=EW)

        def onFind():        # использует поле ввода из внешней обл. видимости
            self.onFind(entry1.get())  # вызов обработчика диалога поиска

        def onApply():
            self.onDoChange(entry1.get(), entry2.get())

        Button(new, text='Find', command=onFind ).grid(row=0,
                                                       column=2, sticky=EW)
        Button(new, text='Apply', command=onApply).grid(row=1,
                                                        column=2, sticky=EW)
        new.columnconfigure(1, weight=1)       # растягиваемые поля ввода

    def onDoChange(self, findtext, changeto):
        # реализует замену для диалога поиска с заменой:
        # заменяет и повторяет поиск
        if self.text.tag_ranges(SEL):          # сначала найти
            self.text.delete(SEL_FIRST, SEL_LAST)
```

```
            self.text.insert(INSERT, changeto) # удалит, если пусто
            self.text.see(INSERT)
            self.onFind(findtext)              # переход к следующему
            self.text.update()                 # принудительное обновление

    def onGrep(self):
        """
        новое в версии 2.1: многопоточная реализация поиска во внешних файлах;
        выполняет поиск указанной строки в файлах, имена которых соответствуют
        заданному шаблону; щелчок на элементе в списке открывает соответствующий
        файл, при этом выполняется переход к строке с найденным вхождением;

        поиск выполняется в отдельном потоке, чтобы графический интерфейс
        не блокировался и оставался активным и чтобы позволить одновременно
        выполнять несколько операций поиска; можно было бы использовать модуль,
        если прекращать цикл проверки при отсутствии активных операций поиска;

        алгоритм выбора кодировки при выполнении поиска: содержимое текстовых
        файлов в дереве, где выполняется поиск, может храниться в любых
        кодировках: мы не предлагаем вводить имя кодировки для каждого файла
        (как при открытии), однако позволяем указать кодировку для всего
        дерева, предварительно устанавливая общесистемную кодировку по
        умолчанию, используемую файловой системой или для представления
        текста, и пропускаем файлы, декодирование которых терпит неудачу;
        в самом тяжелом случае пользователю может потребоваться выполнить
        поиск N раз, если в дереве могут присутствовать файлы с текстом
        в N различных кодировках; иначе операция открытия будет возбуждать
        исключение, а открытие в двоичном режиме может не дать совпадения
        кодированного текста с испытуемой строкой;

        TBD: может, лучше было бы выводить сообщение об ошибке при встрече
        с файлом, который не удалось декодировать?
        но файлы с кодировкой utf-16 (2 байта на символ), созданные в Notepad,
        благополучно могут декодироваться с применением кодировки
        utf-8, однако строка при этом не будет найдена;
        TBD: можно было бы позволить вводить несколько имен кодировок, отделяя
        их друг от друга запятыми, и пробовать применять их поочередно
        к каждому файлу, помимо loadEncode
        """
        from PP4E.Gui.ShellGui.formrows import makeFormRow

        # немодальный диалог: ввод имени каталога, шаблон имени файла,
        # искомая строка
        popup = Toplevel()
        popup.title('PyEdit - grep')
        var1 = makeFormRow(popup, label='Directory root', width=18,
                           browse=False)
        var2 = makeFormRow(popup, label='Filename pattern', width=18,
                           browse=False)
        var3 = makeFormRow(popup, label='Search string', width=18,
                           browse=False)
```

```
        var4 = makeFormRow(popup, label='Content encoding', width=18,
                           browse=False)
        var1.set('.')                       # текущий каталог
        var2.set('*.py')                    # начальные значения
        var4.set(sys.getdefaultencoding()) # для содержимого файлов, а не имен
        cb = lambda: self.onDoGrep(var1.get(), var2.get(), var3.get(),
                                   var4.get())
        Button(popup, text='Go',command=cb).pack()

    def onDoGrep(self, dirname, filenamepatt, grepkey, encoding):
        """
        вызывается щелчком на кнопке Go в диалоге Grep: заполняет список
        найденными совпадениями
        tbd: возможно, следует запускать поток-производитель как демон, чтобы
        он автоматически завершался вместе с приложением?
        """
        import threading, queue

        # создать немодальный и незакрываемый диалог
        mypopup = Tk()
        mypopup.title('PyEdit - grepping')
        status = Label(mypopup,
                       text='Grep thread searching for: %r...' % grepkey)
        status.pack(padx=20, pady=20)
        mypopup.protocol('WM_DELETE_WINDOW', lambda: None) # игнорировать
                                                           # кнопку X
        # запустить поток-производитель, цикл проверки результатов
        myqueue = queue.Queue()
        threadargs = (filenamepatt, dirname, grepkey, encoding, myqueue)
        threading.Thread(target=self.grepThreadProducer,
                         args=threadargs).start()
        self.grepThreadConsumer(grepkey, encoding, myqueue, mypopup)

    def grepThreadProducer(self, filenamepatt, dirname, grepkey, encoding,
                           myqueue):
        """
        выполняется в параллельном потоке, не имеющем отношения к графическому
        интерфейсу: помещает в очередь список с результатами find.find;
        найденные совпадения можно было бы помещать в очередь по мере их
        обнаружения, но для этого необходимо обеспечить сохранение окна на
        экране; здесь могут возникать ошибки декодирования не только
        содержимого, но и имен файлов;

        TBD: чтобы избежать ошибок декодирования имен файлов в
        os.walk/listdir, можно было бы передавать методу find() строку bytes,
        но какую кодировку использовать: sys.getfilesystemencoding(), если она
        не равна None? Смотрите также примечание в разделе "Модуль fnmatch"
        в главе 6: в версии 3.1 модуль fnmatch всегда преобразует текст
        в двоичное представление, используя кодировку Latin-1;
        """
        from PP4E.Tools.find import find
```

```
        matches = []
        try:
            for filepath in find(pattern=filenamepatt, startdir=dirname):
                try:
                    textfile = open(filepath, encoding=encoding)
                    for (linenum, linestr) in enumerate(textfile):
                        if grepkey in linestr:
                            msg = '%s@%d [%s]' % (filepath,
                                                  linenum + 1, linestr)
                            matches.append(msg)
                except UnicodeError as X:                   # напр.: декодир.,
                    print('Unicode error in:', filepath, X) # двоичный режим
                except IOError as X:
                    print('IO error in:', filepath, X) # напр.: права доступа
        finally:
            myqueue.put(matches) # остановить цикл потребителя при исключении:
                                 # имена файлов?
    def grepThreadConsumer(self, grepkey, encoding, myqueue, mypopup):
        """
        выполняется в главном потоке графического интерфейса: просматривает
        очередь в ожидании результатов или []; может иметься несколько
        активных потоков/циклов/очередей, связанных с поиском; в процессе
        могут присутствовать другие типы потоков/циклов проверки, особенно
        если PyEdit прикрепляется как компонент (PyMailGUI);
        """
        import queue
        try:
            matches = myqueue.get(block=False)
        except queue.Empty:
            myargs = (grepkey, encoding, myqueue, mypopup)
            self.after(250, self.grepThreadConsumer, *myargs)
        else:
            mypopup.destroy() # закрыть информационный диалог
            self.update()     # и стереть его с экрана
            if not matches:
                showinfo('PyEdit', 'Grep found no matches for: %r' % grepkey)
            else:
                self.grepMatchesList(matches, grepkey, encoding)

    def grepMatchesList(self, matches, grepkey, encoding):
        """
        заполняет список найденными совпадениями в случае успеха;
        так как поиск увенчался успехом, кодировка уже известна: использовать
        ее в обработчике щелчка на файле в списке, чтобы обеспечить его
        открытие без обращения к пользователю;
        """
        from PP4E.Gui.Tour.scrolledlist import ScrolledList
        print('Matches for %s: %s' % (grepkey, len(matches)))

        # перехватывает двойной щелчок на списке
        class ScrolledFilenames(ScrolledList):
```

```
            def runCommand(self, selection):
                file, line = selection.split(' [', 1)[0].split('@')
                editor = TextEditorMainPopup(
                    loadFirst=file, winTitle=' grep match',
                    loadEncode=encoding)
                editor.onGoto(int(line))
                editor.text.focus_force() # на самом деле не требуется

        # новое модальное окно
        popup = Tk()
        popup.title('PyEdit - grep matches: %r (%s)' % (grepkey, encoding))
        ScrolledFilenames(parent=popup, options=matches)

    ###########################################################################
    # Операции меню Tools
    ###########################################################################
    def onFontList(self):
        self.fonts.append(self.fonts[0]) # выбрать следующий шрифт в списке
        del self.fonts[0]                # изменит размер текстовой области
        self.text.config(font=self.fonts[0])

    def onColorList(self):
        self.colors.append(self.colors[0]) # выбрать следующий цвет в списке
        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 выбора цвета

    def pickColor(self, part): # это очень просто
        (triple, hexstr) = askcolor()
        if hexstr:
            self.text.config(**{part: hexstr})

    def onInfo(self):
        """
        диалог с информацией о тексте и о местоположении курсора;
        ВНИМАНИЕ (2.1): при вычислении позиции курсора библиотека Tk
        считает символ табуляции, как один символ: следует умножать их на 8,
        чтобы обеспечить соответствие с визуальным положением?
        """
        text = self.getAllText()        # добавлено 5/3/00 за 15 мин.
        bytes = len(text)               # словами считается все, что
        lines = len(text.split('\n'))   # отделяется пробелами
        words = len(text.split())       # 3.x: в bytes - символы
        index = self.text.index(INSERT) # в str - кодовые пункты Юникода
        where = tuple(index.split('.'))
        showinfo('PyEdit Information',
```

```
                'Current location:\n\n' +
                'line:\t%s\ncolumn:\t%s\n\n' % where +
                'File text statistics:\n\n' +
                'chars:\t%d\nlines:\t%d\nwords:\t%d\n' % (bytes, lines,
                                                         words))
    def onClone(self, makewindow=True):
        """
        открывает новое окно редактора, не изменяя уже открытое (onNew);
        наследует поведение операции Quit и других от окна, копия которого
        создается;
        2.1: подклассы должны переопределять/замещать этот метод, если
        будут создавать собственные окна,
        иначе этот метод создаст дополнительное поддельное пустое окно;
        """
        if not makewindow:
            new = None              # предполагается, что класс создает
        else:                       # собственное окно
            new = Toplevel()        # новое окно редактора в том же процессе
        myclass = self.__class__ # объект класса экземпляра (самый нижний)
        myclass(new)             # прикрепить/запустить экземпляр моего класса

    def onRunCode(self, parallelmode=True):
        """
        выполнение редактируемого программного кода Python -- это не IDE,
        но удобно; пытается выполнить в каталоге файла, не в cwd (может быть
        корнем PP4E); вводит и добавляет аргументы командной строки для файлов
        сценариев;

        stdin/out/err для программного кода = стартовое окно редактора, если
        оно есть: запускайте редактор в окне консоли, чтобы увидеть вывод,
        производимый программным кодом; если parallelmode=True, открывает окно
        DOS для операций ввода-вывода; путь поиска модулей будет включать '.'
        при запуске; при выполнении программного кода как отдельной строки
        корневым окном может быть окно PyEdit; здесь также можно использовать
        модули subprocess и multiprocessing;

        2.1: исправлено на использование базового имени файла после chdir,
             без пути;
        2.1: использует StartArgs для передачи аргументов в режиме запуска
             файлов в Windows;
        2.1: вызывает update() после первого диалога, в противном случае
             второй диалог иногда не появляется на экране;
        """
        def askcmdargs():
            return askstring('PyEdit', 'Commandline arguments?') or ''

        from PP4E.launchmodes import System, Start, StartArgs, Fork
        filemode = False
        thefile = str(self.getFileName())
        if os.path.exists(thefile):
```

```
            filemode = askyesno('PyEdit', 'Run from file?')
            self.update()                           # 2.1: вызывает update()
        if not filemode:                            # выполнить как строку
            cmdargs = askcmdargs()
            namespace = {'__name__': '__main__'}    # выполнить как сценарий
            sys.argv = [thefile] + cmdargs.split()  # можно использов. потоки
            exec(self.getAllText() + '\n', namespace)# игнорировать исключения
        elif self.text_edit_modified():             # 2.0: проверка изменений
            showerror('PyEdit', 'Text changed: you must save before run')
        else:
            cmdargs = askcmdargs()
            mycwd = os.getcwd()                       # cwd может быть корнем
            dirname, filename = os.path.split(thefile) # каталог, базовое имя
            os.chdir(dirname or mycwd)                # cd для файлов
            thecmd = filename + ' ' + cmdargs         # 2.1: не theFile
            if not parallelmode:                      # выполнить как файл
                System(thecmd, thecmd)()              # блокировать редактор
            else:
                if sys.platform[:3] == 'win':         # породить параллельно
                    run = StartArgs if cmdargs else Start  # 2.1: аргументы
                    run(thecmd, thecmd)()             # или всегда Spawn
                else:
                    Fork(thecmd, thecmd)()            # породить параллельно
            os.chdir(mycwd)                           # вернуться в каталог

    def onPickFont(self):
        """
        2.0 немодальный диалог выбора шрифта
        2.1: поля ввода диалога передаются обработчику, допускается открывать
        одновременно несколько диалогов поиска выбора шрифта
        """
        from PP4E.Gui.ShellGui.formrows import makeFormRow
        popup = Toplevel(self)
        popup.title('PyEdit - font')
        var1 = makeFormRow(popup, label='Family', browse=False)
        var2 = makeFormRow(popup, label='Size', browse=False)
        var3 = makeFormRow(popup, label='Style', browse=False)
        var1.set('courier')
        var2.set('12')          # предлагаемые значения
        var3.set('bold italic') # смотрите допустимые значения в списке выбора
        Button(popup, text='Apply', command=
               lambda: self.onDoFont(var1.get(), var2.get(),
                                     var3.get())).pack()

    def onDoFont(self, family, size, style):
        try:
            self.text.config(font=(family, int(size), style))
        except:
            showerror('PyEdit', 'Bad font specification')
```

```
##############################################################################
# Прочие утилиты, полезные за пределами этого класса
##############################################################################

def isEmpty(self):
    return not self.getAllText()

def getAllText(self):
    return self.text.get('1.0', END+'-1c') # извлечь текст как строку str

def setAllText(self, text):
    """
    вызывающий: должен предварительно вызвать self.update(), если только
    что был прикреплен, иначе начальная позиция может оказаться не в
    первой, а во второй строке (2.1; ошибка Tk?)
    """
    self.text.delete('1.0', END)      # записать текстовую строку в виджет
    self.text.insert(END, text)       # или '1.0'; текст = bytes или str
    self.text.mark_set(INSERT, '1.0') # переместить точку ввода в начало
    self.text.see(INSERT)             # прокрутить в начало, в точку вставки
def clearAllText(self):
    self.text.delete('1.0', END)    # очистить текст в виджете

def getFileName(self):
    return self.currfile
def setFileName(self, name): # смотрите также: onGoto(linenum)
    self.currfile = name     # для последующего сохранения
    self.filelabel.config(text=str(name))

def setKnownEncoding(self, encoding='utf-8'): # 2.1: для сохранения
    self.knownEncoding = encoding # иначе будут использованы настройки,
                                  # запрос?
def setBg(self, color):
    self.text.config(bg=color)    # для установки вручную из программы
def setFg(self, color):
    self.text.config(fg=color)    # 'black', шестнадцатеричная строка
def setFont(self, font):
    self.text.config(font=font)   # ('семейство', размер, 'стиль')

def setHeight(self, lines):       # по умолчанию = 24 строки x 80 символов
    self.text.config(height=lines)# можно также взять из textConfig.py
def setWidth(self, chars):
    self.text.config(width=chars)

def clearModified(self):
    self.text.edit_modified(0)       # сбросить флаг наличия изменений
def isModified(self):                # были изменения с момента
    return self.text_edit_modified() # последнего сброса флага?

def help(self):
    showinfo('About PyEdit', helptext % ((Version,)*2))
```

```
##############################################################################
# Готовые к употреблению классы редактора, подмешиваемые в подкласс
# фрейма GuiMaker, создающий меню и панели инструментов.
#
# Эти классы реализуют типичные случаи использования, однако возможны и другие
# реализации; для запуска PyEdit, как самостоятельной программы, следует
# вызвать метод TextEditorMain().mainloop(); переопределяйте/расширяйте
# в подклассах метод onQuit, чтобы обеспечить перехват события завершения
# приложения или уничтожения окна (смотрите пример PyView);
# ВНИМАНИЕ: можно было бы использовать windows.py для создания ярлыков,
# но здесь используется собственный протокол завершения.
##############################################################################

#-----------------------------------------------------------------------------
# 2.1: в quit(), не завершать без предупреждения, если в процессе открыты
# другие окна редактора и в них имеются несохраненные изменения - изменения
# будут потеряны, потому что все остальные окна тоже закрываются, включая
# множественные родительские окна Tk, включающие редактор; для слежения за
# всеми окнами PyEdit используется список экземпляров, созданных в процессе;
# это может оказаться чрезмерной мерой (если вместо quit() вызывается
# destroy(), когда достаточно проверить только дочернее окно редактирования
# уничтожаемого родителя), но лучше перестраховаться; метод onQuit перемещен
# сюда, потому что его реализация отличается для окон разных типов и может
# присутствовать не во всех окнах;
#
# предполагается, что TextEditorMainPopup никогда не будет играть роль
# родителя для других окон редактирования - дочерние виджеты Toplevel
# уничтожаются вместе со своими родителями; это не позволяет предотвратить
# закрытие из-за пределов классов PyEdit (метод quit в tkinter доступен
# во всех виджетах, и любой виджет может быть родителем для Toplevel!);
# ответственность за проверку наличия изменений в содержимом редактора
# полностью возлагается на клиента; обратите внимание, что в данной ситуации
# привязка события <Destroy> не даст ровным счетом ничего, потому что его
# обработчик не может выполнять операции с графическим интерфейсом, такие как
# проверка наличия изменений и извлечение текста, - дополнительную информацию
# об этом событии смотрите в книге и в модуле destroyer.py;
#-----------------------------------------------------------------------------

########################################
# когда текстовый редактор владеет окном
########################################

class TextEditorMain(TextEditor, GuiMakerWindowMenu):
    """
    главное окно редактора PyEdit, которое вызывает метод quit() при
    выполнении операции Quit графического интерфейса для завершения
    приложения и конструирует меню в окне; родителем может быть окно Tk,
    по умолчанию, окно Tk, создаваемое явно, или объект Toplevel:
    родитель должен быть окном и, вероятно, окном Tk, чтобы избежать закрытия
    без предупреждения вместе с родителем; при выполнении операции Quit
    графического интерфейса все главные окна PyEdit проверяют остальные окна
```

```
    PyEdit, открытые в процессе, на наличие несохраненных изменений, поскольку
    вызов метода quit() здесь приведет к завершению всего приложения; фрейм
    редактора необязательно должен занимать окно целиком (окно может включать
    и другие компоненты: смотрите PyView), но его операция Quit завершает
    программу; метод onQuit вызывается операцией Quit, выполняемой щелчком на
    кнопке в панели инструментов, выбором пункта в меню File, а также щелчком
    на кнопке X в заголовке окна;
    """
    def __init__(self, parent=None, loadFirst='', loadEncode=''):
        # редактор занимает все родительское окно
        GuiMaker.__init__(self, parent)        # использует главное меню окна
        TextEditor.__init__(self, loadFirst, loadEncode)# фрейм GuiMaker
                                                        # прикрепляет себя сам
        self.master.title('PyEdit ' + Version) # заголовок, кнопка X, если
        self.master.iconname('PyEdit')         # выполняется как отдельная
        self.master.protocol('WM_DELETE_WINDOW', self.onQuit) # программа
        TextEditor.editwindows.append(self)

    def onQuit(self):                          # вызывается операцией Quit
        close = not self.text_edit_modified() # проверить себя, запросить,
        if not close:                          # проверить другие
            close = askyesno('PyEdit',
                             'Text changed: quit and discard changes?')
        if close:
            windows = TextEditor.editwindows
            changed = [w for w in windows
                       if w != self and w.text_edit_modified()]
            if not changed:
                GuiMaker.quit(self) # завершить все приложение, независимо от
            else:                   # типа виджета
                numchange = len(changed)
                verify = '%s other edit window%s changed: '
                verify = verify + 'quit and discard anyhow?'
                verify = verify % (numchange, 's' if numchange > 1 else '')
                if askyesno('PyEdit', verify):
                    GuiMaker.quit(self)

class TextEditorMainPopup(TextEditor, GuiMakerWindowMenu):
    """
    всплывающее окно PyEdit, которое вызывает метод destroy() при выполнении
    операции Quit графического интерфейса, закрывает только себя и создает
    меню в окне; создает собственного родителя Toplevel, который является
    дочерним для окна Tk по умолчанию (если передается значение None) или для
    другого указанного окна или виджета (например, для фрейма);
    добавляется в список для проверки при закрытии любого главного окна
    PyEdit; если будет создано главное окно PyEdit, родитель данного окна
    также должен быть родителем главного окна PyEdit, чтобы оно не было
    закрыто без предупреждения; метод onQuit вызывается операцией Quit,
    выполняемой щелчком на кнопке в панели инструментов, выбором пункта в меню
    File, а также щелчком на кнопке X в заголовке окна;
    """
```

```
    def __init__(self, parent=None, loadFirst='', winTitle='', loadEncode=''):
        # создать собственное окно
        self.popup = Toplevel(parent)
        GuiMaker.__init__(self, self.popup) # использует главное меню окна
        TextEditor.__init__(self, loadFirst, loadEncode) # фрейм в новом окне
        assert self.master == self.popup
        self.popup.title('PyEdit ' + Version + winTitle)
        self.popup.iconname('PyEdit')
        self.popup.protocol('WM_DELETE_WINDOW', self.onQuit)
        TextEditor.editwindows.append(self)

    def onQuit(self):
        close = not self.text_edit_modified()
        if not close:
            close = askyesno('PyEdit',
                             'Text changed: quit and discard changes?')
        if close:
            self.popup.destroy()                  # закрыть только это окно
            TextEditor.editwindows.remove(self)   # (и все дочерние окна)

    def onClone(self):
        TextEditor.onClone(self, makewindow=False) # я создаю собственное окно

##############################################
# когда редактор встраивается в другое окно
##############################################

class TextEditorComponent(TextEditor, GuiMakerFrameMenu):
    """
    прикрепляемый фрейм компонента PyEdit с полными меню/панелью
    инструментов, который вызывает destroy() при выполнении операции Quit
    графического интерфейса и стирает только себя; при выполнении операции
    Quit проверяется наличие несохраненных изменений только в этом редакторе;
    не перехватывает щелчок на кнопке X в заголовке окна: не имеет
    собственного окна; не добавляет себя в список отслеживаемых окон: является
    частью более крупного приложения;
    """
    def __init__(self, parent=None, loadFirst='', loadEncode=''):
        # использовать меню на основе фрейма
        GuiMaker.__init__(self, parent) # все меню, кнопки в GuiMaker должны
        TextEditor.__init__(self, loadFirst, loadEncode) # создаваться первыми

    def onQuit(self):
        close = not self.text_edit_modified()
        if not close:
            close = askyesno('PyEdit',
                             'Text changed: quit and discard changes?')
        if close:
            self.destroy() # стереть свой фрейм, но не завершать вмещающее
                           # приложение
class TextEditorComponentMinimal(TextEditor, GuiMakerFrameMenu):
```

```
    """
    прикрепляемый фрейм компонента PyEdit без операции Quit и без меню File;
    на запуске удаляет кнопку Quit из панели инструментов и удаляет меню
    File или запрещает все его пункты (грубовато, зато эффективно);
    структуры меню и панели инструментов являются данными экземпляра:
    изменение их не затрагивает другие экземпляры;
    Операция Quit графического интерфейса никогда не запускается, потому что
    она удаляется из доступных операций;
    """
    def __init__(self, parent=None, loadFirst='',
                 deleteFile=True, loadEncode=''):
        self.deleteFile = deleteFile
        GuiMaker.__init__(self, parent) # фрейм GuiMaker прикрепляет себя сам
        TextEditor.__init__(self, loadFirst, loadEncode) # TextEditor
                                                         # добавляется
                                                         # в середину
    def start(self):
        TextEditor.start(self)               # вызов метода start GuiMaker
        for i in range(len(self.toolBar)):   # удалить quit из панели инстр.
            if self.toolBar[i][0] == 'Quit': # удалить пункты меню file
                del self.toolBar[i]          # или просто запретить их
                break
        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])

################################################################################
# запуск как самостоятельной программы
################################################################################

def testPopup():
    # проверку запуска как компонента смотрите 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():                 # из командной строки или щелчком
    try:                    # либо как ассоциированная программа в Windows
        fname = sys.argv[1] # аргумент = необязательное имя файла
    except IndexError:      # создается в корневом окне Tk по умолчанию
```

```
        fname = None
    TextEditorMain(loadFirst=fname).pack(expand=YES, fill=BOTH)# pack -
    mainloop()                                                  # необязательно

if __name__ == '__main__':  # когда запускается как сценарий
    #testPopup()
    main()                  # используйте .pyw, чтобы запустить без окна DOS
```

PyPhoto: программа просмотра и изменения размеров изображений

В главе 9 мы написали простую программу просмотра миниатюр изображений, реализующую прокрутку коллекции миниатюр в холсте. Эта программа в свою очередь была основана на приемах и программном коде для работы с изображениями, разработанных в конце главы 8. В обоих случаях я обещал, что мы в конечном счете встретимся с более полнофункциональным воплощением рассматривавшихся идей.

В этом разделе мы наконец завершим обсуждение миниатюр изображений знакомством с PyPhoto – улучшенной программой просмотра и изменения размеров изображений. Основу программы PyPhoto составляют простые операции: для заданного каталога с изображениями PyPhoto отображает их миниатюры в холсте с прокруткой. При выборе миниатюры отображается соответствующее ей полноразмерное изображение во всплывающем окне.

В отличие от предыдущих программ просмотра изображений, PyPhoto предусматривает возможность прокрутки изображения (вместо обрезания), если оно оказывается больше физического экрана. Кроме того, программа PyPhoto вводит понятие изменения размеров изображения – она поддерживает события от мыши и клавиатуры, которые изменяют размер изображения по одной из осей и увеличивают или уменьшают масштаб изображения. После того как изображение открыто, логика изменения размеров позволяет растягивать и сжимать изображение до произвольных размеров, что особенно удобно при просмотре цифровых фотографий, которые могут быть слишком большими, чтобы их можно было просмотреть целиком.

Кроме того, PyPhoto позволяет сохранять изображения в файлах (возможно, после изменения размеров) и дает возможность выбирать и открывать каталоги с изображениями в самом графическом интерфейсе, а не только с помощью аргументов командной строки.

Все вместе особенности PyPhoto образуют программу обработки изображений, хотя и с небольшим, по современным меркам, количеством инструментов. Я предлагаю вам самим попробовать добавить в нее новые возможности – после овладения навыками работы с прикладным интерфейсом библиотеки Python Imaging Library (PIL) объектно-

ориентированная природа PyPhoto делает добавление новых инструментов удивительно простым делом.

Запуск PyPhoto

Чтобы запустить PyPhoto, необходимо получить и установить пакет расширения PIL, описанный в главе 8. Программа PyPhoto использует многие функциональные возможности PIL; эта библиотека поддерживает дополнительные форматы изображений, помимо тех, что поддерживаются стандартной библиотекой tkinter (например, изображений JPEG), и используется для выполнения операций над изображениями, таких как изменение размеров, создание миниатюр и сохранение в файлах. Расширение PIL распространяется с открытыми исходными текстами, как и Python, но в настоящее время оно не является частью стандартной библиотеки Python. Ищите PIL в Интернете (в настоящее время самый точный адрес: *http://www.pythonware.com*). Проверьте также каталог *Extensions* в дереве примеров, где находится самоустанавливающийся пакет PIL.

Самый лучший способ получить представление о программе PyPhoto – запустить ее у себя на компьютере и посмотреть, как она выполняет прокрутку изображений и изменение их размеров. Ниже будет представлено несколько снимков с экрана, дающих общее представление о взаимодействии с программой. Запустить PyPhoto можно щелчком на ее ярлыке или из командной строки. При непосредственном запуске программа открывает подкаталог *images* в исходном каталоге, который содержит несколько фотографий. При запуске из командной строки программе можно передать имя начального каталога в виде аргумента. На рис. 11.7 изображено главное окно с миниатюрами, которое выводится при непосредственном запуске программы.

Прежде чем появится это окно, PyPhoto загружает или создает миниатюры, используя инструменты, реализованные в главе 8. Если каталог с изображениями открывается впервые, запуск программы может занять несколько секунд, но все последующие запуски будут протекать быстро – PyPhoto кэширует миниатюры в локальном подкаталоге, чтобы можно было пропустить этап создания миниатюр, когда этот же каталог будет открыт в следующий раз.

Технически, существует три разных варианта поведения PyPhoto при запуске: она отображает содержимое определенного каталога, указанного в командной строке; отображает содержимое каталога *images* по умолчанию при запуске без аргументов командной строки и когда каталог *images* находится в каталоге запуска программы; или отображает единственную кнопку, после щелчка на которой предоставляется возможность выбрать и открыть каталог для просмотра, когда начальный каталог не указан или отсутствует (смотрите логику работы раздела `__main__`).

Рис. 11.7. *Главное окно PyPhoto, каталог по умолчанию*

Программа PyPhoto позволяет также открывать дополнительные папки в новых окнах с миниатюрами, для чего достаточно нажать клавишу `D` в окне с миниатюрами или в окне с изображением. Например, на рис. 11.8 показан диалог выбора новой папки с изображениями в Windows 7, а на рис. 11.9 показан результат того, что я открыл каталог, куда были скопированы фотографии с карты памяти моей цифровой фотокамеры, – это второе окно PyPhoto с миниатюрами на экране. Окно, изображенное на рис. 11.8, также открывается из окна с единственной кнопкой, если при запуске программе не указать начальный каталог или если этот каталог недоступен.

После выбора миниатюры на экране появляется новое окно, где в холсте отображается соответствующее изображение. Если изображение оказывается слишком большим для экрана, его можно будет прокрутить с помощью полос прокрутки в окне. На рис. 11.10 показано изображение, которое было выведено после щелчка на миниатюре, а на рис. 11.11 – диалог `Save As`, запущенный нажатием клавиши `S` в окне с изображением. В этом диалоге `Save As` необходимо ввести требуемое

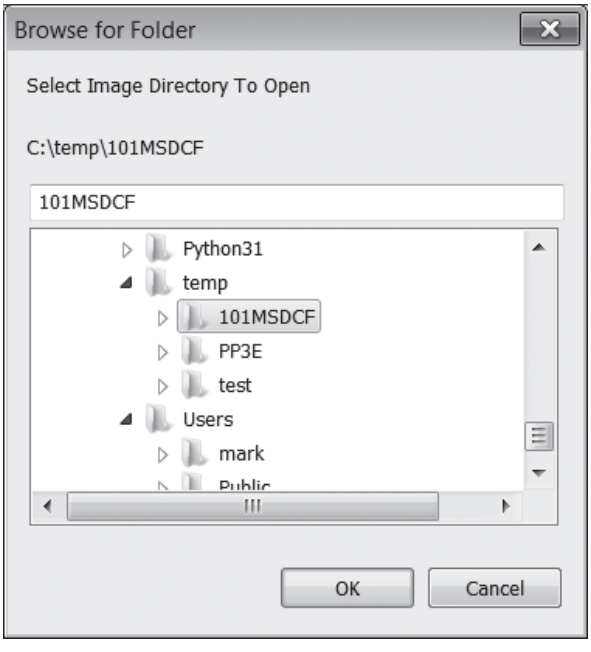


Рис. 11.8. *Диалог открытия каталога в программе PyPhoto (клавиша D)*

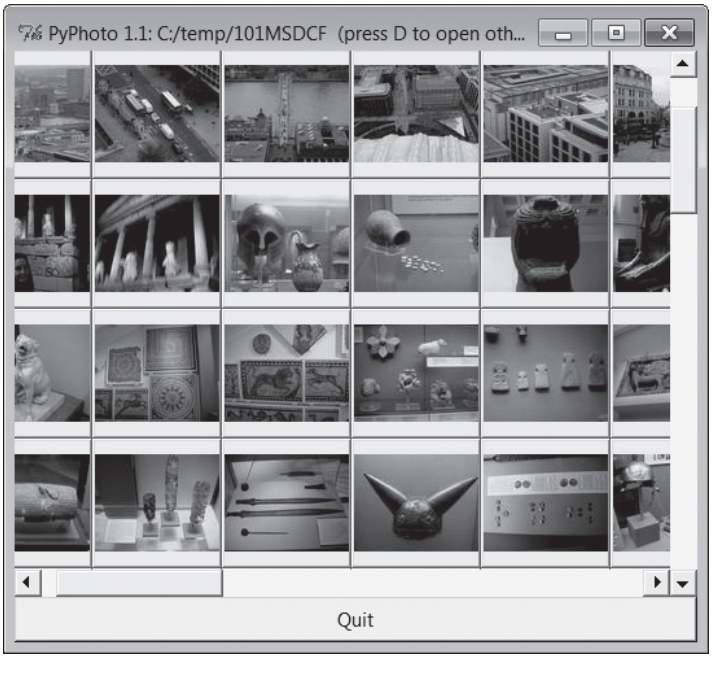


Рис. 11.9. *Окно PyPhoto с миниатюрами, другой каталог*

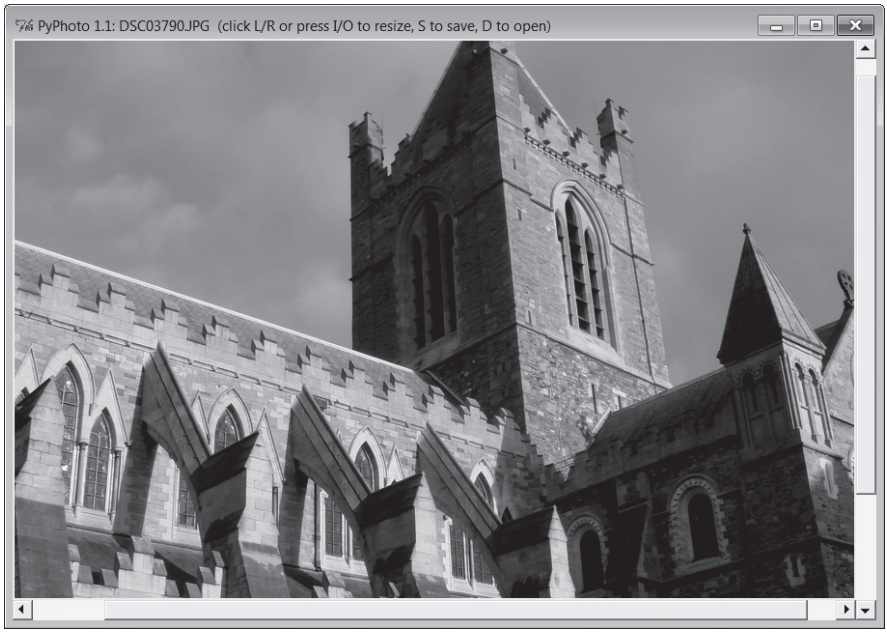


Рис. 11.10. *Окно PyPhoto для просмотра изображения*

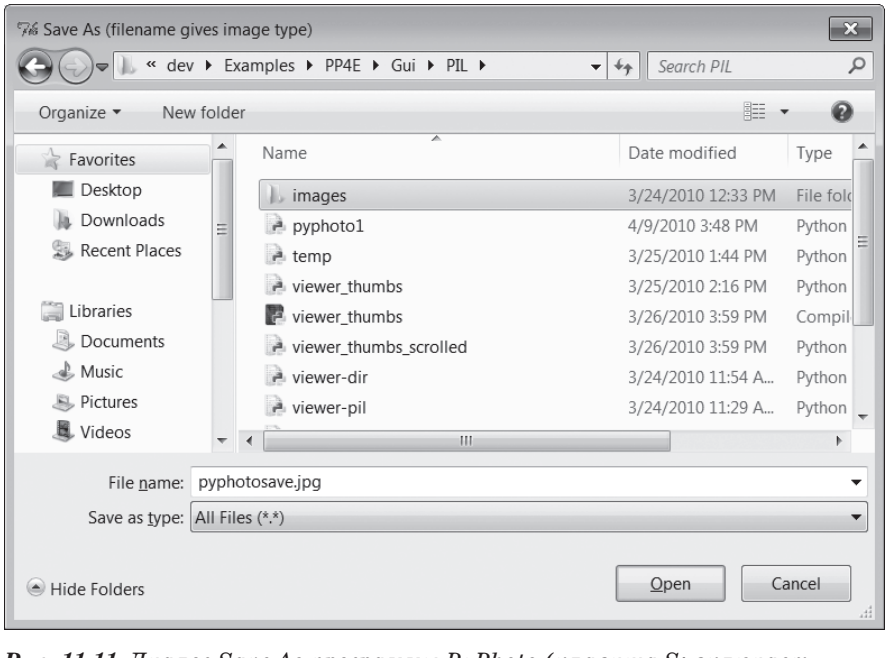


Рис. 11.11. *Диалог Save As программы PyPhoto (клавиша S; включает расширение)*

расширение имени файла (например, *.jpg*), потому что расширение PIL использует его, чтобы определить, в каком формате сохранять изображение в файле. В целом, программа PyPhoto позволяет открывать любое количество окон с миниатюрами и с полноразмерными изображениями, и каждое изображение может сохраняться независимо от других.

Помимо уже показанных снимков с экранов, довольно сложно изобразить особенности взаимодействий с программой в такой статической среде, как книга, – более полное представление вы сможете получить, опробовав программу у себя на компьютере.

Например, щелчки левой и правой кнопками мыши будут изменять высоту и ширину изображения, соответственно, а нажатие клавиш I и 0 будет изменять масштаб, увеличивая и уменьшая изображение с шагом 10 процентов. Обе схемы изменения размеров позволяют сжать изображение, которое не умещается на экране целиком, а также растягивать маленькие фотографии. Они также сохраняют исходное отношение сторон фотографий, пропорционально изменяя высоту или ширину, чего не делает изменение размера только по одной из осей (может растягиваться по ширине или высоте).

После изменения размеров изображения можно сохранять в файлах с их текущими размерами. Кроме того, программа PyPhoto достаточно интеллектуальна, чтобы открывать в Windows окна в полный размер, если изображение не помещается в открытое окно.

Исходный программный код PyPhoto

Поскольку PyPhoto просто расширяет и повторно использует приемы и программный код, с которыми мы встречались ранее в книге, здесь мы опустим детальное обсуждение исходных текстов. За исходными сведениями обращайтесь к обсуждению приемов обработки изображений и применения PIL в главе 8 и к описанию виджета холста в главе 9.

В двух словах отмечу, что PyPhoto использует холсты в двух случаях: для отображения коллекций миниатюр и для вывода открываемых изображений. Для вывода миниатюр используется тот же прием компоновки, что и раньше, в примере 9.15. Для вывода полноразмерных изображений также используется холст, прокручиваемая (полная) область которого соответствует размеру изображения, а видимая область вычисляется как минимум из размера физического экрана и размера самого изображения. Физический размер экрана можно определить вызовом метода `maxsize()` окна `Toplevel`. Благодаря этому полноразмерное изображение можно прокручивать, что очень удобно при просмотре изображений, размеры которых слишком велики, чтобы уместиться на экране (что весьма характерно для фотографий, снятых новейшими цифровыми фотокамерами).

Кроме того, PyPhoto выполняет привязку событий от клавиатуры и мыши для реализации операций изменения размеров и масштаби-

рования. Благодаря PIL эти операции реализуются очень просто – мы сохраняем оригинальное изображение в объекте изображения PIL, вызываем его метод `resize`, передавая новые размеры, и перерисовываем изображение на холсте. Программа PyPhoto также использует диалоги открытия и сохранения файла, чтобы запомнить последний посещенный каталог.

Расширение PIL поддерживает дополнительные операции, которыми мы могли бы расширить набор обрабатываемых событий, но для просмотра изображений вполне достаточно изменения размеров. В настоящее время PyPhoto не использует потоки выполнения, чтобы с их помощью избежать блокирования во время выполнения продолжительных операций (например, операция первого открытия большого каталога). Такие расширения я оставляю для самостоятельного упражнения.

Программа PyPhoto реализована в виде единого файла, представленного в примере 11.5, хотя она получает бесплатно некоторую дополнительную функциональность от повторного использования функции, генерирующей миниатюры, из модуля `viewer_thumbs`, который мы написали в конце главы 8, в примере 8.45. Чтобы не заставлять вас листать страницы взад и вперед, ниже приводится фрагмент программного кода импортируемой функции создания миниатюр, используемой здесь:

```
# импортировано из главы 8...

def makeThumbs(imgdir, size=(100, 100), subdir='thumbs'):
    # возвращает список кортежей
    # (имя_файла_изображения, объект_миниатюры_изображения);
    thumbdir = os.path.join(imgdir, subdir)
    if not os.path.exists(thumbdir):
        os.mkdir(thumbdir)

    thumbs = []
    for imgfile in os.listdir(imgdir):
        thumbpath = os.path.join(thumbdir, imgfile)
        if os.path.exists(thumbpath):
            thumbobj = Image.open(thumbpath) # использовать созданные ранее
            thumbs.append((imgfile, thumbobj))
        else:
            print('making', thumbpath)
            imgpath = os.path.join(imgdir, imgfile)
            try:
                imgobj = Image.open(imgpath)            # создать миниатюру
                imgobj.thumbnail(size, Image.ANTIALIAS) # фильтр, дающий
                                                        # лучшее качество при
                                                        # уменьшении размеров
                imgobj.save(thumbpath)                  # тип определяется
                thumbs.append((imgfile, imgobj))        # расширением
            except:
                print("Skipping: ", imgpath)
    return thumbs
```

Программный код, реализующий окно выбора миниатюр, также очень схож с представленным в главе 9 примером с прокручиваемой коллекцией миниатюр, но он не импортируется этим файлом, а просто повторяется в нем, чтобы обеспечить будущее его развитие (и его статус в главе 9 – функционального подмножества – здесь понижен до уровня прототипа).

При изучении этого файла особое внимание обратите на организацию программного кода в виде набора функций и методов многократного пользования, которая позволяет избежать избыточности, – если нам, например, когда-нибудь придется изменить реализацию операции изменения размеров, нам достаточно будет изменить один метод, а не два. Кроме того, обратите внимание на класс `ScrolledCanvas` – компонент многократного пользования, который обеспечивает автоматическое связывание полос прокрутки и холстов.

Пример 11.5. PP4E\Gui\PIL\pyphoto1.py

```
"""
###############################################################################
PyPhoto 1.1: программа просмотра миниатюр изображений с возможностью изменения
размеров и сохранения.

Позволяет открывать несколько окон для просмотра миниатюр из разных каталогов
- в качестве начального каталога с изображениями принимается аргумент командной
строки, каталог по умолчанию "images" или выбранный щелчком на кнопке в главном
окне; последующие каталоги могут открываться нажатием клавиши "D" в окне
с миниатюрами или в окне просмотра полноразмерного изображения.

Программа также позволяет прокручивать изображения, если они слишком большие
и не умещаются на экране;
все еще необходимо: (1) реализовать переупорядочение миниатюр при изменении
размеров окна, исходя из текущего размера окна; (2) [ВЫПОЛНЕНО] возможность
изменения размеров изображения в соответствии с текущими размерами окна?
(3) отключать прокрутку, если размер изображения меньше максимального размера
окна: использовать Label, если шир_изобр <= шир_окна и выс_изобр <= выс_окна?

Новое в версии 1.1: работает под управлением Python 3.1 и с последней версией PIL;

Новое в версии 1.0: реализован пункт (2) выше: щелчок мышью изменяет размер
изображения в соответствии с одним из размеров экрана, и предусмотрена
возможность увеличения и уменьшения масштаба изображения с шагом 10% нажатием
клавиши; требуется поискать более универсальные решения; ВНИМАНИЕ: похоже, что
после многократного изменения размеров теряется качество изображения (вероятно,
это ограничение PIL)

Следующий алгоритм масштабирования, заимствованный из реализации создания
миниатюр средствами PIL, напоминает алгоритм масштабирования по высоте экрана,
используемый в программе, но только для сжатия:
x, y = imgwide, imghigh
if x > scrwide: y = max(y * scrwide // x, 1); x = scrwide
```

```
if y > scrhigh: x = max(x * scrhigh // y, 1); y = scrhigh
##############################################################################
"""

import sys, math, os
from tkinter import *
from tkinter.filedialog import SaveAs, Directory

from PIL import Image                # PIL Image: также имеется в tkinter
from PIL.ImageTk import PhotoImage   # версия виджета PhotoImage из PIL
from viewer_thumbs import makeThumbs # разработан ранее в книге

# запомнить последний открытый каталог
saveDialog = SaveAs(title='Save As (filename gives image type)')
openDialog = Directory(title='Select Image Directory To Open')

trace = print # or lambda *x: None
appname = 'PyPhoto 1.1: '

class ScrolledCanvas(Canvas):
    """
    холст в контейнере, который автоматически создает вертикальную и
    горизонтальную полосы прокрутки
    """
    def __init__(self, container):
        Canvas.__init__(self, container)
        self.config(borderwidth=0)
        vbar = Scrollbar(container)
        hbar = Scrollbar(container, orient='horizontal')

        vbar.pack(side=RIGHT, fill=Y)        # холст прикрепляется после
        hbar.pack(side=BOTTOM, fill=X)       # полос, чтобы обрезался первым
        self.pack(side=TOP, fill=BOTH, expand=YES)

        vbar.config(command=self.yview)      # вызвать при перемещении полосы
        hbar.config(command=self.xview)      # прокрутки
        self.config(yscrollcommand=vbar.set) # вызвать при прокрутке холста
        self.config(xscrollcommand=hbar.set)

class ViewOne(Toplevel):
    """
    при создании открывает единственное изображение во всплывающем окне;
    реализовано в виде класса, потому что объект PhotoImage должен
    сохраняться, иначе изображение будет стерто при утилизации; обеспечивает
    прокрутку больших изображений; щелчок мыши изменяет размер изображения
    в соответствии с высотой или шириной окна: растягивает или сжимает; нажатие
    клавиш I и O увеличивает и уменьшает размеры изображения; оба алгоритма
    изменения размеров предусматривают сохранение оригинального отношения
    сторон; программный код организован так, чтобы избежать избыточности,
    насколько это возможно;
    """
```

```
def __init__(self, imgdir, imgfile, forcesize=()):
    Toplevel.__init__(self)
    helptxt = '(click L/R or press I/O to resize, S to save, D to open)'
    self.title(appname + imgfile + ' ' + helptxt)
    imgpath = os.path.join(imgdir, imgfile)
    imgpil = Image.open(imgpath)
    self.canvas = ScrolledCanvas(self)
    self.drawImage(imgpil, forcesize)
    self.canvas.bind('<Button-1>', self.onSizeToDisplayHeight)
    self.canvas.bind('<Button-3>', self.onSizeToDisplayWidth)
    self.bind('<KeyPress-i>',      self.onZoomIn)
    self.bind('<KeyPress-o>',      self.onZoomOut)
    self.bind('<KeyPress-s>',      self.onSaveImage)
    self.bind('<KeyPress-d>',      onDirectoryOpen)
    self.focus()

def drawImage(self, imgpil, forcesize=()):
    imgtk = PhotoImage(image=imgpil)              # file != imgpath
    scrwide, scrhigh = forcesize or self.maxsize() # размеры x,y экрана
    imgwide  = imgtk.width()                      # размеры в пикселях
    imghigh  = imgtk.height()                     # то же,
                                                  # что и imgpil.size
    fullsize = (0, 0, imgwide, imghigh)           # прокручиваемая
    viewwide = min(imgwide, scrwide)              # видимая
    viewhigh = min(imghigh, scrhigh)

    canvas = self.canvas
    canvas.delete('all')                          # удалить предыд. изобр.
    canvas.config(height=viewhigh, width=viewwide) # видимые размеры окна
    canvas.config(scrollregion=fullsize)          # размер прокр. области
    canvas.create_image(0, 0, image=imgtk, anchor=NW)

    if imgwide <= scrwide and imghigh <= scrhigh:  # слишком велико?
        self.state('normal')                  # нет: размер окна по изобр.
    elif sys.platform[:3] == 'win':           # в Windows на весь экран
        self.state('zoomed')                  # в других исп. geometry()
    self.saveimage = imgpil
    self.savephoto = imgtk                    # сохранить ссылку на меня
    trace((scrwide, scrhigh), imgpil.size)

def sizeToDisplaySide(self, scaler):
    # изменить размер, чтобы полностью заполнить одну сторону экрана
    imgpil = self.saveimage
    scrwide, scrhigh = self.maxsize() # размеры x,y экрана
    imgwide, imghigh = imgpil.size    # размеры изображения в пикселях
    newwide, newhigh = scaler(scrwide, scrhigh, imgwide, imghigh)
    if (newwide * newhigh < imgwide * imghigh):
        filter = Image.ANTIALIAS      # сжатие: со сглаживанием
    else:                             # растягивание: бикубическая
        filter = Image.BICUBIC        # аппроксимация
```

```
        imgnew = imgpil.resize((newwide, newhigh), filter)
        self.drawImage(imgnew)

    def onSizeToDisplayHeight(self, event):
        def scaleHigh(scrwide, scrhigh, imgwide, imghigh):
            newhigh = scrhigh
            newwide = int(scrhigh * (imgwide / imghigh)) # истинное деление
            return (newwide, newhigh)                    # пропорциональные
        self.sizeToDisplaySide(scaleHigh)

    def onSizeToDisplayWidth(self, event):
        def scaleWide(scrwide, scrhigh, imgwide, imghigh):
            newwide = scrwide
            newhigh = int(scrwide * (imghigh / imgwide)) # истинное деление
            return (newwide, newhigh)
        self.sizeToDisplaySide(scaleWide)

    def zoom(self, factor):
        # уменьшить или увеличить масштаб с шагом
        imgpil = self.saveimage
        wide, high = imgpil.size
        if factor < 1.0:             # сглаживание дает лучшее качество
            filter = Image.ANTIALIAS # при сжатии, также можно
        else:                        # использовать NEAREST, BILINEAR
            filter = Image.BICUBIC
        new = imgpil.resize((int(wide * factor), int(high * factor)), filter)
        self.drawImage(new)

    def onZoomIn(self, event, incr=.10):
        self.zoom(1.0 + incr)

    def onZoomOut(self, event, decr=.10):
        self.zoom(1.0 - decr)

    def onSaveImage(self, event):
        # сохранить изображение в текущем виде в файл
        filename = saveDialog.show()
        if filename:
            self.saveimage.save(filename)

    def onDirectoryOpen(event):
        """
        открывает новый каталог с изображениями в новом окне
        может вызываться в обоих окнах, с изображением и с миниатюрами
        """
        dirname = openDialog.show()
        if dirname:
            viewThumbs(dirname, kind=Toplevel)

    def viewThumbs(imgdir, kind=Toplevel, numcols=None,
                   height=400, width=500):
```

```
    """
    создает окно и кнопки с миниатюрами;
    использует кнопки фиксированного размера, прокручиваемый холст;
    устанавливает прокручиваемый (полный) размер и размещает миниатюры
    в холсте по абсолютным координатам x,y;
    больше не предполагает, что все миниатюры имеют одинаковые размеры:
    за основу берет максимальные размеры (x,y) среди всех миниатюр,
    некоторые могут быть меньше;
    """
    win = kind()
    helptxt = '(press D to open other)'
    win.title(appname + imgdir + ' ' + helptxt)
    quit = Button(win, text='Quit', command=win.quit, bg='beige')
    quit.pack(side=BOTTOM, fill=X)
    canvas = ScrolledCanvas(win)
    canvas.config(height=height, width=width) # видимый размер окна, может
                                              # изменяться пользователем
    thumbs = makeThumbs(imgdir)               # [(imgfile, imgobj)]
    numthumbs = len(thumbs)
    if not numcols:
        numcols = int(math.ceil(math.sqrt(numthumbs))) # фиксир. или N x N
    numrows = int(math.ceil(numthumbs / numcols))      # истинное деление

    # максимальная шир|выс: thumb=(name, obj), thumb.size=(width, height)
    linksize = max(max(thumb[1].size) for thumb in thumbs)
    trace(linksize)
    fullsize = (0, 0,                             # X,Y верхн. левого угла
         (linksize*numcols),(linksize*numrows))  # X,Y прав. нижнего угла
    canvas.config(scrollregion=fullsize)         # размер прокруч. области

    rowpos = 0
    savephotos = []
    while thumbs:
        thumbsrow, thumbs = thumbs[:numcols], thumbs[numcols:]
        colpos = 0
        for (imgfile, imgobj) in thumbsrow:
            photo = PhotoImage(imgobj)
            link  = Button(canvas, image=photo)
            def handler(savefile=imgfile):
                ViewOne(imgdir, savefile)
            link.config(command=handler, width=linksize, height=linksize)
            link.pack(side=LEFT, expand=YES)
            canvas.create_window(colpos, rowpos, anchor=NW,
                        window=link, width=linksize, height=linksize)
            colpos += linksize
            savephotos.append(photo)
        rowpos += linksize
    win.bind('<KeyPress-d>', onDirectoryOpen)
    win.savephotos = savephotos
    return win
```

```
if __name__ == '__main__':
    """
    открываемый каталог = по умолчанию или из аргумента командной строки,
    иначе вывести простое окно с кнопкой для выбора каталога
    """
    imgdir = 'images'
    if len(sys.argv) > 1: imgdir = sys.argv[1]
    if os.path.exists(imgdir):
        mainwin = viewThumbs(imgdir, kind=Tk)
    else:
        mainwin = Tk()
        mainwin.title(appname + 'Open')
        handler = lambda: onDirectoryOpen(None)
        Button(mainwin, text='Open Image Directory', command=handler).pack()
    mainwin.mainloop()
```

PyView: слайд-шоу для изображений и примечаний

Одна картинка стоит тысячи слов, и их понадобится значительно меньше, чтобы вывести картинку с помощью Python. В следующей программе, PyView, представлена простая и переносимая реализация алгоритма слайд-шоу на языке Python и в библиотеке tkinter. Эта программа не обладает возможностями обработки изображений, такими как изменение их размеров, но она реализует другие инструменты, такие как файлы с примечаниями для изображений, и может выполняться при отсутствии PIL.

Запуск PyView

В PyView соединились многие из тем, изучавшихся в главе 9: последовательная смена изображений реализована с применением метода `after`, объекты изображений выводятся на холсте, автоматически изменяющем размер, и так далее. В главном окне программы на холсте выводится фотография; пользователь может открыть и просматривать ее непосредственно или запустить режим поочередного показа слайдов, в котором фотографии, случайным образом выбранные из каталога, выводятся через равные промежутки времени, задаваемые с помощью виджета ползунка.

По умолчанию показ слайдов в PyView производится для каталога с изображениями, входящего в состав примеров для книги (хотя кнопка `Open` позволяет загружать изображения из любых каталогов). Чтобы посмотреть другую коллекцию фотографий, передайте имя каталога в качестве первого аргумента командной строки или измените имя каталога по умолчанию в самом сценарии. Я не могу показать, как действует программа в режиме показа слайдов, но главное окно привести можно.

На рис. 11.12 изображено главное окно PyView, созданное сценарием *slideShowPlus.py* из примера 11.6, как оно выглядит в Windows 7.

На рисунке в книге 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», чтобы определить отсутствие задержки между сменой фотографий, и щелкните на 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 применить где-либо, кроме как в рекламе, действующей на подсознание. Демонстрируемые фотографии загружаются при начальном запуске, чтобы сохранить ссылки на них (напомню, что объекты с изображениями нужно удерживать). Но скорость, с которой могут отображаться большие GIF-файлы на языке Python, впечатляет, а то и просто восхищает.

Во время показа слайдов кнопка Start изменяется на Stop (изменяется ее текстовый атрибут с помощью метода `config` виджета). На рис. 11.13 изображено окно после щелчка на кнопке Stop в некоторый момент.

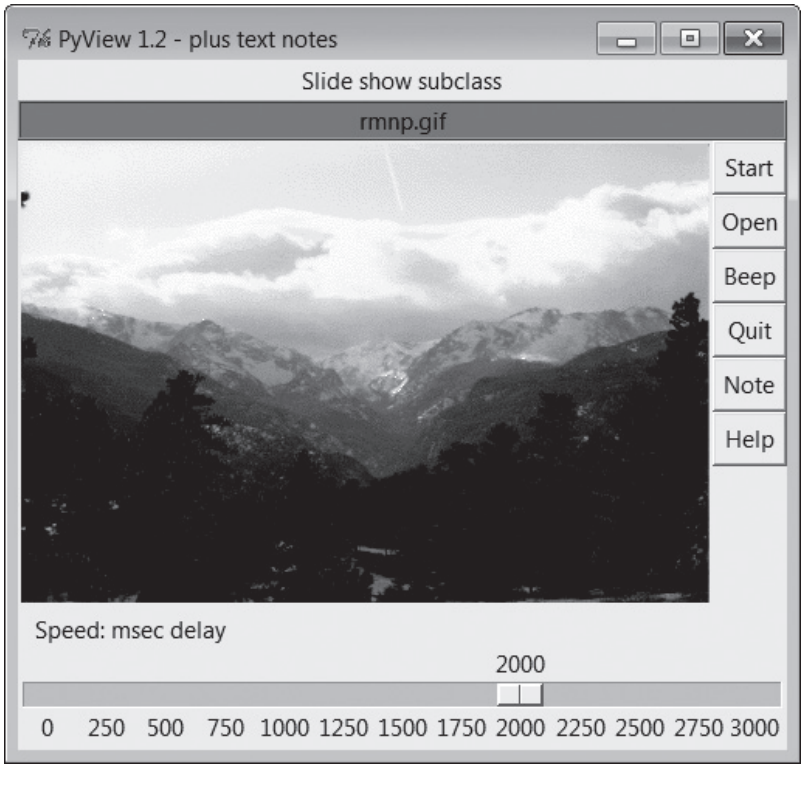


Рис. 11.12. *PyView без примечаний*

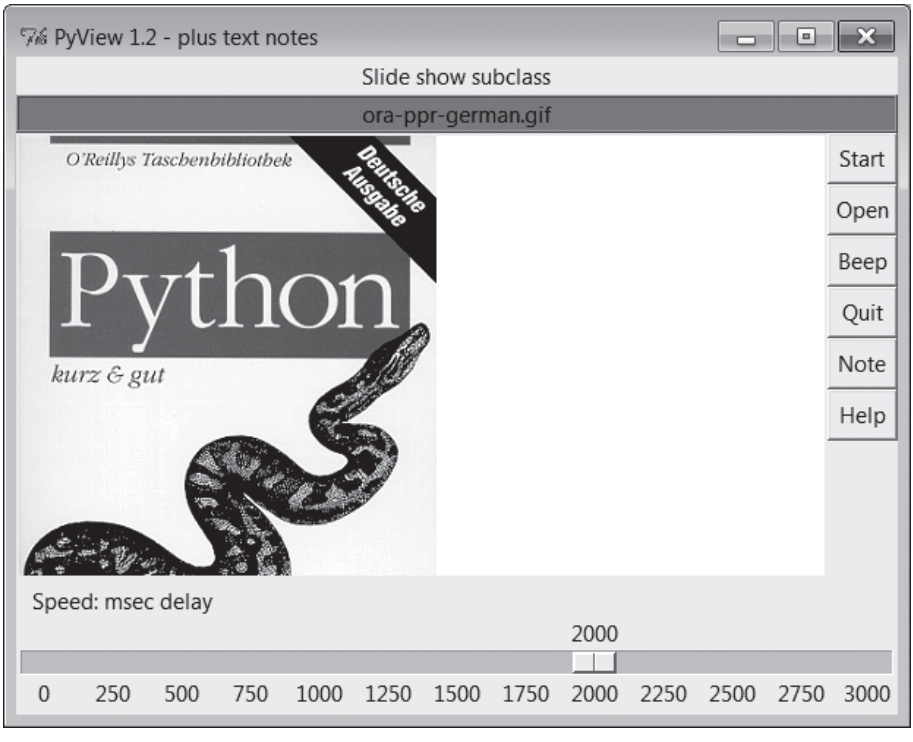


Рис. 11.13. *PyView после остановки показа слайдов*

Кроме того, у каждой фотографии может быть свой файл «примечаний», который автоматически открывается вместе с изображением. С помощью этой функции можно записывать основные данные о фотографии. Нажмите кнопку Note, чтобы открыть дополнительный набор виджетов, с помощью которых можно просматривать и изменять файл примечаний, связанный с фотографией, просматриваемой в данный момент. Этот дополнительный набор виджетов должен показаться вам знакомым – это текстовый редактор PyEdit, представленный ранее в этой главе, прикрепленный к PyView в качестве средства просмотра и редактирования примечаний к фотографиям. На рис. 11.14 показано окно программы PyView вместе с прикрепленным к нему компонентом PyEdit для редактирования примечаний.

Встраивание PyEdit в PyView

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 был приведен выше. Так как редактор PyEdit

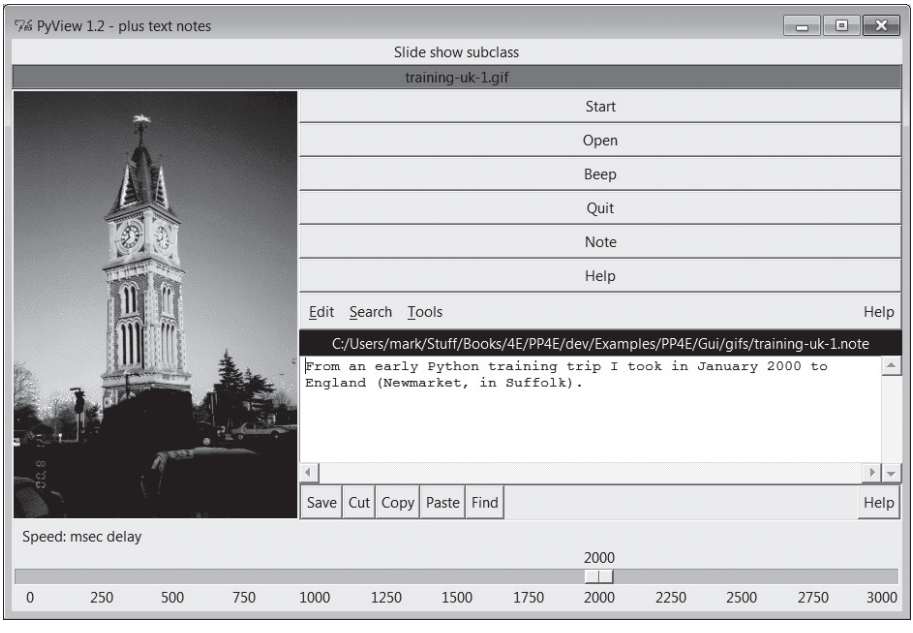


Рис. 11.14. *PyView с примечаниями*

реализован в виде класса, подобным образом его можно повторно использовать в любом графическом интерфейсе, где требуется обеспечить возможность редактирования текста.

При встраивании таким способом PyEdit оказывается вложенным фреймом, прикрепляемым к фрейму в интерфейсе программы слайд-шоу. При этом PyEdit создает меню на основе фрейма (он не владеет окном в целом), текстовое содержимое сохраняется и выбирается непосредственно вмещающей программой, а некоторые возможности автономного режима отсутствуют (например, отсутствуют меню File и кнопка Quit). При этом вы получаете все остальные функции PyEdit, включая вырезание и копирование, поиск и поиск с заменой, поиск во внешних файлах, настройку цвета и шрифта, поддержку отмены и возврата операций редактирования и так далее. Доступна даже операция Clone, которая создает новое окно редактирования, хотя при этом меню создается на основе фрейма без операции Quit и без меню File, а при выходе не проверяется наличие несохраненных изменений, – все эти функции при желании можно связать с новым классом компонента PyEdit верхнего уровня.

Кроме того, если передать PyView третий аргумент командной строки, после имени каталога с изображениями, он будет интерпретироваться как индекс в списке классов PyEdit в соответствии с режимами верхнего уровня. Значению 0 аргумента соответствует режим главного окна,

в этом случае редактор примечаний помещается под изображением, а его меню – в верхнюю часть окна (его фрейм при компоновке получает оставшееся место в окне, а не во фрейме PyView). При значении 1 редактор выводится в отдельном, независимом окне `Toplevel` (деактивируется при выключении показа примечаний). При значениях 2 и 3 PyEdit используется как встраиваемый компонент, прикрепляемый к фрейму PyView, с меню на основе фрейма (при значении 2 редактор включает все имеющиеся у него пункты меню, которые могут не подходить для данного случая его применения, а значение 3 обеспечивает ограниченный набор пунктов меню).

На рис. 11.15 изображен случай использования значения 0, когда PyEdit запускается в режиме главного окна. Здесь в окне в действительности создаются два независимых фрейма – фрейм PyView в верхней части и фрейм текстового редактора в нижней части. Недостаток этого режима перед режимом вложенного компонента или отдельного окна состоит в том, что PyEdit берет управление окном программы на

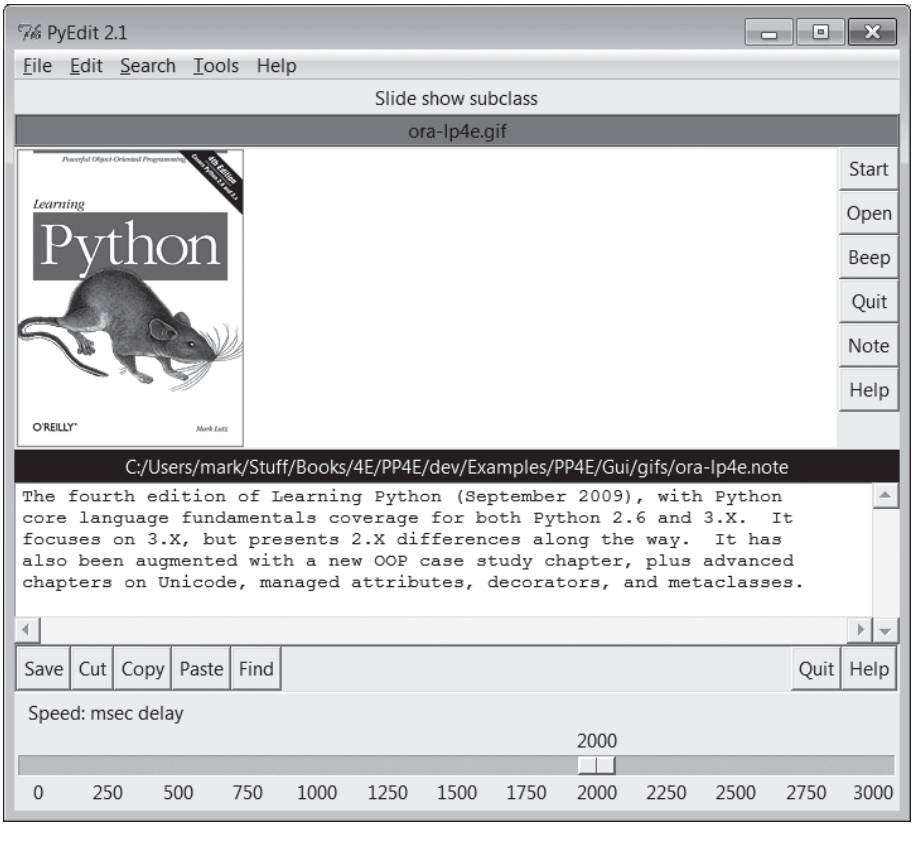


Рис. 11.15. *PyView с PyEdit в другом режиме*

себя (включая его заголовок и обработку события щелчка на кнопке закрытия), а его расположение в нижней части окна означает, что редактор может оказаться скрытым при просмотре изображений большого размера по высоте. Поэкспериментируйте с этой возможностью у себя, чтобы почувствовать особенности использования других разновидностей PyEdit, используя командную строку такого вида:

```
C:\...\PP4E\Gui\SlideShow> slideShowPlus.py ../gifs 0
```

Средство просмотра примечаний появляется только после щелчка на кнопке Note и удаляется после повторного щелчка на ней. Чтобы показать или скрыть фрейм просмотра примечаний, PyView пользуется методами `pack` и `pack_forget` виджетов, с которыми мы познакомились в конце главы 9. Окно автоматически расширяется, чтобы разместить средство просмотра примечаний, когда оно прикрепляется и отображается. Очень важно, что при переводе в видимое состояние редактор повторно прикрепляется с параметрами `expand=YES` и `fill=BOTH`, иначе в некоторых режимах он не будет растягиваться – фрейм PyEdit компонует себя в `GuiMaker` именно с этим параметрами, когда создается впервые, но метод `pack_forget`, похоже... действительно забывает[1].

Файл примечаний можно также открыть во всплывающем окне PyEdit, но по умолчанию PyView встраивает редактор, чтобы сохранить прямую зрительную ассоциацию между изображением и примечанием и избежать проблем, которые могут возникнуть при независимом закрытии окна редактора. В данной реализации классы PyEdit приходится обертывать классом `WrapEditor`, чтобы перехватить операцию уничтожения фрейма PyEdit, когда он выполняется в отдельном всплывающем окне или в режиме полнофункционального компонента, – после уничтожения редактор будет недоступен, и его невозможно будет вновь прикрепить к графическому интерфейсу. Это не представляет проблемы при использовании редактора в режиме главного окна (операция Quit завершает программу) или в режиме минимального компонента (когда редактор не имеет операции Quit). Мы еще встретимся с приемом встраивания PyEdit внутрь другого графического интерфейса, когда будем рассматривать PyMailGUI в главе 14.

Предупреждение: в таком виде PyView поддерживает те же форматы представления графических изображений, что и объект PhotoImage библиотеки tkinter, поэтому по умолчанию он ищет файлы в формате GIF. Улучшить положение можно, установив расширение PIL для просмотра JPEG (и многих других форматов). Поскольку сегодня PIL является необязательным расширением, он не включен в данную версию PyView. Подробнее о расширении PIL и о графических форматах рассказывается в конце главы 8.

---

[1] Здесь игра слов: имя метода `pack_forget` дословно означает «забыть, что был скомпонован». – *Прим. перев.*

Исходный программный код PyView

Поскольку программа PyView разрабатывалась поэтапно, вам придется изучить объединение двух файлов и классов, чтобы понять, как она в действительности работает. 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 11.6 добавляет ряд функций в импортируемый базовый класс показа слайдов – редактирование примечаний, ползунок, определяющий задержку, метку для отображения имени файла и так далее. Это тот файл, который фактически запускает PyView.

Пример 11.6. PP4E\Gui\SlideShow\slideShowPlus.py

```
"""
###############################################################################
PyView 1.2: программа показа слайдов с прилагаемыми к ним примечаниями.

Подкласс класса SlideShow, который добавляет отображение содержимого файлов
с примечаниями в прикрепляемом объекте PyEdit, ползунок для установки интервала
задержки между сменами изображений и метку с именем текущего отображаемого файла
изображения;

Версия 1.2 работает под управлением Python 3.x и дополнительно использует
улучшенный алгоритм повторного прикрепления компонента PyEdit, чтобы обеспечить
его растягиваемость, перехватывает операцию закрытия примечания в подклассе,
чтобы избежать появления исключения при закрытии PyEdit, использующегося
в режиме всплывающего окна или полнофункционального компонента, и вызывает метод
update() перед вставкой текста во вновь прикрепленный редактор примечаний, чтобы
обеспечить правильное позиционирование в первой строке (смотрите описание этой
проблемы в книге).
###############################################################################
"""

import os
from tkinter import *
from PP4E.Gui.TextEditor.textEditor import *
from slideShow import SlideShow
#from slideShow_threads import SlideShow
Size = (300, 550)   # 1.2: начальные размеры, (высота, ширина)

class SlideShowPlus(SlideShow):
    def __init__(self, parent, picdir, editclass, msecs=2000, size=Size):
        self.msecs = msecs
        self.editclass = editclass
        SlideShow.__init__(self, parent, picdir, msecs, size)

    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)

        # 1.2: знать о закрытии редактора необходимо, если он используется
        # в режиме всплывающего окна или полнофункционального компонента
        self.editorGone = False
        class WrapEditor(self.editclass):# расширяет PyEdit для перехвата Quit
            def onQuit(editor):          # editor - экземпляр PyEdit
                self.editorGone = True   # self - вмещающий экземпляр
                self.editorUp = False    # класса слайд-шоу
                self.editclass.onQuit(editor) # предотвратить рекурсию

        # прикрепить фрейм редактора к окну или к фрейму слайд-шоу
        if issubclass(WrapEditor, TextEditorMain): # создать объект редактора
            self.editor = WrapEditor(self.master)  # указать корень для меню
        else:                                  # встраиваемый компонент
            self.editor = WrapEditor(self)     # или компонент всплывающего окна
        self.editor.pack_forget()              # скрыть редактор при запуске
        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 = int(value)

def onNote(self):
    if self.editorGone:     # 1.2: был уничтожен
        return              # не воссоздавать: видимо, он был нежелателен
    if self.editorUp:
        #self.saveNote()           # если редактор уже открыт
        self.editor.pack_forget()   # сохранить текст?, скрыть редактор
        self.editorUp = False
    else:
        # 1.2: повторно прикрепить с параметрами, управляющими
        #      растягиванием, иначе виджет редактора не будет
        #      растягиваться
        # 1.2: вызвать update после прикрепления и перед вставкой текста,
        #      иначе текстовый курсор будет изначально помещен во 2 строку
        self.editor.pack(side=TOP, expand=YES, fill=BOTH)
        self.editorUp = True # или показать/прикрепить редактор
        self.update()        # смотрите Pyedit: та же проблема с loadFirst
        self.loadNote()      # и загрузить текст примечания

def switchNote(self):
    if self.editorUp:
        self.saveNote()      # сохранить примечание к текущему изображению
        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()  # текст может отсутствовать
        if currfile and currtext:
            try:
                open(currfile, 'w').write(currtext)
            except:
                pass # неудача является нормальным явлением при
                     # выполнении за пределами текущего каталога

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() # примечание может отсутствовать

def onHelp(self):
    showinfo('About PyView',
             'PyView version 1.2\nMay, 2010\n(1.1 July, 1999)\n'
             'An image slide show\nProgramming Python 4E')
```

```
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][int(sys.argv[2])]
        except: pass

    root = Tk()
    root.title('PyView 1.2 - plus text notes')
    Label(root, text="Slide show subclass").pack()
    SlideShowPlus(parent=root, picdir=picdir, editclass=editstyle)
    root.mainloop()
```

Базовая функциональность, расширяемая классом `SlideShowPlus`**, приводится в примере 11.7. Этот пример представляет первоначальную реализацию показа слайдов – он открывает файлы изображений, отображает их и организует показ слайдов в цикле. Его можно запустить как самостоятельный сценарий, но при этом вы не получите дополнительных функций, таких как примечания и ползунки, добавляемые подклассом** `SlideShowPlus`**.**

Пример 11.7. PP4E\Gui\SlideShow\slideShow.py

```
"""
###############################################################################
SlideShow: простая реализация показа слайдов на Python/tkinter;
базовый набор функций, реализованных здесь, можно расширять в подклассах;
###############################################################################
"""

from tkinter import *
from glob import glob
from tkinter.messagebox import askyesno
from tkinter.filedialog import askopenfilename
import random
Size = (450, 450) # начальная высота и ширина холста

imageTypes = [('Gif files', '.gif'), # для диалога открытия файла
              ('Ppm files', '.ppm'), # плюс jpg с исправлениями Tk,
              ('Pgm files', '.pgm'), # плюс растровые с помощью BitmapImage
              ('All files', '*')]
```

```
class SlideShow(Frame):
    def __init__(self, parent=None, picdir='.', msecs=3000, size=Size,**args):
        Frame.__init__(self, parent, **args)
        self.size = size
        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 = [(x, PhotoImage(file=x)) for x in files]
        self.msecs = msecs
        self.beep = True
        self.drawn = None

    def makeWidgets(self):
        height, width = self.size
        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 = True
        self.onoff.config(text='Stop', command=self.onStop)
        self.canvas.config(height=self.size[0], width=self.size[1])
        self.onTimer()

    def onStop(self):
        self.loop = False
        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 = not self.beep # toggle, or use ^ 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 1.2')
    root.iconname('PyView')
    Label(root, text="Python Slide Show Viewer").pack()
    SlideShow(root, picdir=picdir, bd=3, relief=SUNKEN)
    root.mainloop()
```

Чтобы вы могли получить более полное представление о том, что реализует этот базовый класс, на рис. 11.16 показано, как выглядит графический интерфейс, создаваемый этим примером, если запустить его в виде самостоятельного сценария. Здесь изображены два экземпляра, создаваемые сценарием `slideShow_frames`, который можно найти в дереве примеров и основная реализация которого приводится ниже:

```
root = Tk()
Label(root, text="Two embedded slide shows: Frames").pack()
SlideShow(parent=root, picdir=picdir, bd=3, relief=SUNKEN).pack(side=LEFT)
SlideShow(parent=root, picdir=picdir, bd=3, relief=SUNKEN).pack(side=RIGHT)
root.mainloop()
```

Простой сценарий `slideShow_frames` прикрепляет два экземпляра `SlideShow` к одному окну. Это возможно благодаря тому, что информация о состоянии сохраняется не в глобальных переменных, а в переменных экземпляра класса. Сценарий `slideShow_toplevels` (также можно найти в дереве примеров) прикрепляет два экземпляра `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.

Рис. 11.16. *Два прикрепленных объекта SlideShow*

PyDraw: рисование и перемещение графики

В главе 9 мы познакомились с простыми приемами воспроизведения анимации с помощью инструментов из библиотеки tkinter (смотрите версии `canvasDraw` в обзоре tkinter). Представленная здесь программа PyDraw, основываясь на тех же идеях, реализует на языке Python более богатые функциональные возможности. В ней появились новые режимы рисования мышью, возможность заливки объектов и фона, встраивание фотографий и многое другое. Кроме того, в ней реализованы приемы перемещения объектов и анимации – нарисованные объекты можно перемещать по холсту, щелкая на них и перетаскивая мышью, и любой нарисованный объект можно плавно переместить через экран в место, указанное щелчком мыши.

Запуск PyDraw

PyDraw, по сути, представляет собой холст tkinter с многочисленными привязками событий от клавиатуры и мыши, которые дают возможность пользователю осуществлять стандартные операции рисования. Эту программу нельзя назвать графическим редактором профессионального уровня, но поразвлечься с ней можно. На самом деле – даже нужно, поскольку книга не позволяет передать такие вещи, как движущийся объект. Запустите PyDraw из какой-нибудь панели запуска программ (или непосредственно файл *movingpics.py* из примера 11.8). Нажмите клавишу ? и посмотрите подсказку по всем имеющимся командам (или прочтите строку `helpstr` в листинге).

На рис. 11.17 изображено окно PyDraw после того как на холсте было нарисовано несколько объектов. Чтобы переместить какой-либо из объектов, щелкните на нем средней кнопкой мыши и перетащите указателем мыши, либо щелкните средней кнопкой на объекте, а затем правой кнопкой в том месте, куда требуется его переместить. В последнем случае PyDraw воспроизводит анимационный эффект, постепенно перемещая объект в указанное место. Попробуйте сделать это с картинкой, находящейся вверху, и вы увидите, как она плавно перемещается по экрану.

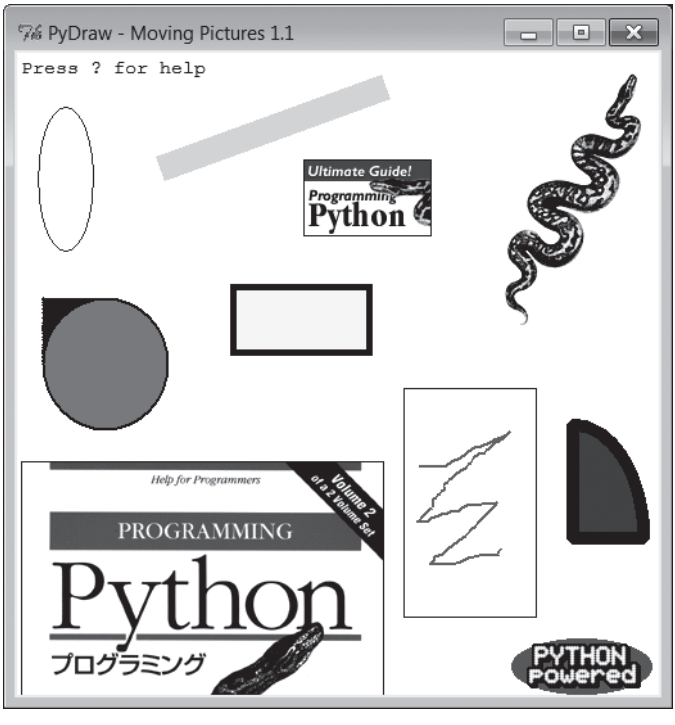


Рис. 11.17. *Окно программы PyDraw с нарисованными объектами, готовыми к перемещению*

Для вставки фотографий нажмите клавишу p, для рисования фигур используйте левую кнопку мыши. (Пользователям Windows: щелчок средней кнопкой обычно равносилен нажатию двух кнопок одновременно или повороту колесика, но для этого может потребоваться выполнить настройки в Панели Управления.) Помимо событий от мыши можно пользоваться еще 17 командами клавиш для редактирования рисунков, о которых я не буду рассказывать здесь. Требуется некоторое время, чтобы освоиться со всеми командами клавиатуры и мыши, после чего вы тоже сможете создавать бессмысленные электронные рисованные объекты, такие как приведены на рис. 11.18.

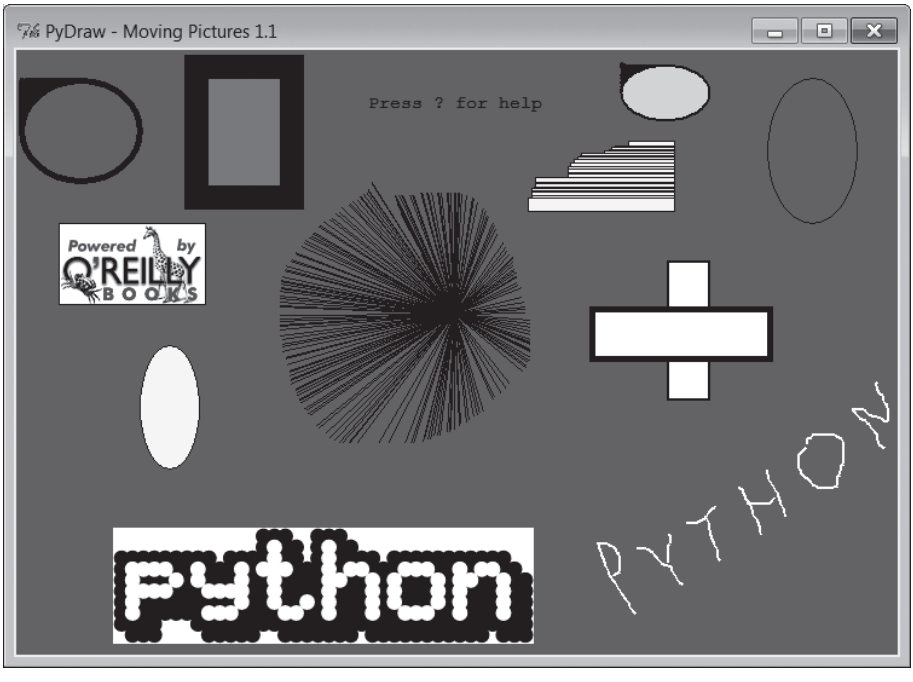


Рис. 11.18. *Окно программы PyDraw после экспериментов с ней*

Исходный программный код PyDraw

Как и PyEdit, программа PyDraw размещается в одном файле. За главным модулем, представленным в примере 11.8, приводятся два расширения, изменяющие реализацию перемещения.

Пример 11.8. PP4E\Gui\MovingPics\movingpics.py

```
"""
###############################################################################
PyDraw 1.1: простая программа рисования на холсте и перемещения объектов
с воспроизведением анимационного эффекта.
В реализации перемещения объектов используются циклы time.sleep, поэтому
в каждый момент времени может перемещаться только один объект; перемещение
выполняется плавно и быстро, однако далее приведены подклассы, реализующие
другие режимы перемещения на основе метода widget.after и потоков выполнения.
Версия 1.1 была дополнена возможностью выполнения под управлением Python 3.X
(версия 2.X не поддерживается)
###############################################################################

"""
helpstr = """--PyDraw версия 1.1--
Операции, выполняемые мышью:
```

```
  Левая                = Начальная точка рисования
  Левая+Перемещение    = Рисовать новый объект
  Двойной щелчок левой = Удалить все объекты
  Правая               = Переместить текущий объект
  Средняя              = Выбрать ближайший объект
  Средняя+Перемещение  = Перетащить текущий объект
Keyboard commands:
  w=Выбрать ширину рамки           c=Выбрать цвет
  u= Выбрать шаг перемещения       s=Выбрать задержку при перемещении
  o=Рисовать овалы                 r=Рисовать прямоугольники
  l=Рисовать линии                 a=Рисовать дуги
  d=Удалить объект                 1=Поднять объект
  2=Опустить объект                f=Выполнить заливку объекта
  b=Выполнить заливку фона         p=Добавить фотографию
  z=Сохранить в формате Postscript x=Выбрать режим рисования
  ?=Справка                        другие=стереть текст
"""

import time, sys
from tkinter import *
from tkinter.filedialog import *
from tkinter.messagebox 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 1.1')
        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     # нач. координаты для следующей итерации

    def onClear(self, event):
        if self.moving: return     # если идет перемещение
        event.widget.delete('all') # использовать тег 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   # план анимированного перемещения
        diffY = event.y - self.where.y   # по горизонтали, затем по вертикали
        reptX = abs(diffX) // pickUnits[0] # приращение на шаге, число шагов
        reptY = abs(diffY) // pickUnits[0] # от предыдущего до текущего щелчка
        incrX = pickUnits[0] * ((diffX > 0) or -1) # 3.x  требуется деление //
        incrY = pickUnits[0] * ((diffY > 0) or -1) # с усечением
        return incrX, reptX, incrY, reptY

    def onMove(self, event):
        traceEvent('onMove', event, 0)     # переместить объект в точку щелчка
        object = self.object                    # игнорировать некоторые
        if object and object not in self.moving: # операции при движении
            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 выполнит другие операции
            time.sleep(pickDelays[0]) # приостановить до следующего шага
        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] # кортеж

def onDrag(self, event):
    diffX = event.x - self.where.x     # OK, если объект перемещается
    diffY = event.y - self.where.y     # переместить в новом направлении
    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 несвязанные методы
       'b': MovingPics.fillBackground, # иначе передавать self в lambda
       '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 объекта 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 перемещается
        self.canvas.delete(self.object) # стереть, но движение продолжится
        self.object = None

def raiseObject(self):
    if self.object:                      # ok если объект перемещается
        self.canvas.tkraise(self.object) # поднять в процессе перемещения

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)       # загрузить изображение
            self.images.append(image)           # сохранить ссылку
            self.object = self.canvas.create_image(         # на холст,
                                self.where.x, self.where.y, # в точку
                                image=image, anchor=NW)     # посл. щелчка

def savePostscript(self):
    file = asksaveasfilename()
    if file:
        self.canvas.postscript(file=file)       # сохранить холст в файл

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() # стандартная операция закрытия окна: сообщит
                            # об ошибке, если выполняется перемещение

    def traceEvent(label, event, fullTrace=True):
        print(label)
        if fullTrace:
            for atrr in dir(event):
                if attr[:2] != '__':
                    print(attr, '=>', getattr(event, attr))

if __name__ == '__main__':
    from sys import argv                # когда выполняется как сценарий,
    if len(argv) == 2: PicDir = argv[1] # '..' не действует при запуске из
                                        # другого каталога
    root = Tk()                         # создать и запустить объект
    MovingPics(root)                    # MovingPics
    root.mainloop()
```

Так как одновременно перемещаться может только один объект, запуск процедуры перемещения объекта в тот момент, когда другой уже находится в движении, приводит к приостановке перемещения первого объекта, пока не будет закончено перемещение нового. Так же как в примерах `canvasDraw` из главы 9, можно добавить поддержку одновременного перемещения более чем одного объекта с помощью событий планируемых обратных вызовов `after` или потоков выполнения.

В примере 11.9 приводится подкласс `MovingPics`, в котором проведены изменения, необходимые для обеспечения параллельного перемещения с помощью событий `after`. Он позволяет одновременно и независимо друг от друга перемещать любое количество объектов на холсте, включая картинки. Запустите этот файл непосредственно, и вы увидите разницу – я мог бы попытаться сделать снимок с экрана в момент, когда одновременно перемещаются несколько объектов, но из этого вряд ли бы что-то вышло.

Пример 11.9. PP4E\Gui\MovingPics\movingpics_after.py

```
"""
PyDraw-after: простая программа рисования на холсте и перемещения объектов
с воспроизведением анимационного эффекта.
Для реализации перемещения объектов используются циклы на основе метода widget.
after, благодаря чему оказалось возможным организовать одновременное перемещение
```

```
нескольких объектов без применения потоков выполнения; движение осуществляется
параллельно, но медленнее, чем в версии с использованием time.sleep; смотрите
также пример canvasDraw в обзоре: он конструирует и передает сразу весь список
incX/incY: здесь могло бы быть 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 -= 1
        else:
            self.canvas.move(objectId, 0, incrY)
            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 как сценарий
    if len(argv) == 2:
        import movingpics           # глобальная перем. не из этого модуля
        movingpics.PicDir = argv[1] # а from* не связывает имена
    root = Tk()
    MovingPicsAfter(root)
    root.mainloop()
```

Чтобы оценить работу этого примера, распахните окно сценария на весь экран и создайте несколько объектов на его холсте, нажимая клавишу p после предварительного щелчка, чтобы вставить картинки, нарисуйте несколько фигур и так далее. Теперь, когда уже выполняется одно или несколько перемещений, можно запустить перемещение еще одного объекта, щелкнув на нем средней кнопкой и затем правой кнопкой

в том месте, куда требуется его переместить. Перемещение начинается немедленно, даже если на холсте присутствуют другие движущиеся объекты. Запланированные события `after` всех объектов помещаются в одну и ту же очередь цикла событий и передаются библиотекой tkinter после срабатывания таймера настолько быстро, насколько возможно.

Если запустить этот модуль подкласса непосредственно, то можно заметить, что перемещение не такое плавное и быстрое, как первоначально (в зависимости от быстродействия вашего компьютера и наличия дополнительных программных уровней под Python), зато одновременно может выполняться несколько перемещений.

В примере 11.10 демонстрируется, как обеспечить параллельное перемещение нескольких объектов с помощью потоков. Этот прием действует, но, как отмечалось в главах 9 и 10, обновление графического интерфейса в дочерних потоках выполнения является, вообще говоря, опасным делом. На моей машине перемещение в этом сценарии с потоками происходит не так плавно, как в первоначальной версии, что отражает накладные расходы, связанные с переключением интерпретатора (и ЦП) между несколькими потоками, но, опять же, во многом это зависит от быстродействия компьютера.

Пример 11.10. PP4E\Gui\MovingPics\movingpics_threads.py

```
"""
PyDraw-threads: использует потоки для перемещения объектов; прекрасно работает
в Windows, если не вызывать метод canvas.update() в потоках (иначе сценарий
будет завершаться с фатальными ошибками, некоторые объекты будут начинать
движение сразу после того как будут нарисованы, и так далее); имеется
как минимум несколько методов холста, которые могут вызываться из потоков
выполнения; движение осуществляется менее плавно, чем с применением time.sleep,
и данная реализация более опасна в целом: внутри потоков лучше ограничиться
изменением глобальных переменных и никак не касаться графического интерфейса;
"""
import _thread as 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 виртуальной машины: не поможет
    def onMove(self, event):
        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])  # приостановиться до следующего шага
        #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: виджет аналоговых/цифровых часов

Изучая новый интерфейс компьютера, я всегда вначале отыскиваю часы. Я столько времени неотрывно нахожусь за компьютером, что у меня совершенно не получается следить за временем, если оно не отображается прямо передо мной на экране (и даже тогда это проблематично). Следующая программа, PyClock, реализует такой виджет часов на языке Python. Своим внешним видом она не очень отличается от тех часов, которые вы привыкли видеть в системе X Window. Но так как она написана на языке Python, ее легко перенастраивать и переносить между Windows, X Window и Mac, как и все программы из этой главы. В дополнение к развитым технологиям конструирования графических интерфейсов, этот пример демонстрирует использование модулей Python `math` и `time`.

Краткий урок геометрии

Прежде чем продемонстрировать вам PyClock, немного предыстории и признаний. Ну-ка, ответьте: как поставить точки на окружности? Эта задача, а также форматы времени и возникающие события оказываются основными при создании графических элементов часов. Чтобы нарисовать циферблат аналоговых часов на холсте, необходимо уметь рисовать круг – сам циферблат состоит из точек окружности, а секундная,

минутная и часовая стрелки представляют собой линии, проведенные из центра в точки на окружности. Цифровые часы нарисовать проще, но смотреть на них неинтересно.

Теперь признание: начав писать PyClock, я не знал ответа на первый вопрос предыдущего абзаца. Я совершенно забыл формулу нахождения координат точек окружности (как и большинство профессиональных программистов, к которым я с этим обращался). Бывает. Такие знания, не будучи востребованными в течение нескольких десятилетий, могут быть утилизированы сборщиком мусора. В конце концов мне удалось смахнуть пыль с нескольких нейронов, длины которых оказалось достаточно, чтобы запрограммировать действия, необходимые для построения, но блеснуть умом мне не удалось.[1]

Если вы в таком же положении, то я покажу вам один способ простой записи формул построения точек на языке Python, хотя для подробных занятий геометрией места здесь нет. Прежде чем взяться за более сложную задачу реализации часов, я написал сценарий `plotterGui`, представленный в примере 11.11, чтобы сосредоточиться только на логике построения круга.

Логика построения круга реализуется в функции `point` – она находит координаты (X,Y) точки окружности по относительному номеру точки, общему количеству точек, помещаемых на окружности, и радиусу окружности (расстоянию между центром окружности и ее точками). Сначала вычисляется угол между нужной точкой и верхней точкой окружности путем деления 360 на количество рисуемых точек и умножения на номер точки. Напомню, что полный круг составляет 360 градусов (например, если на окружности рисуется 4 точки, то каждая отстоит от предыдущей на 90 градусов, или на 360/4). Стандартный модуль Python `math` предоставляет все необходимые константы и функции – `pi`, `sine` и `cosine`. В действительности математика тут не такая уж непонятная, если вы потратите некоторое время, чтобы ее рассмотреть (возможно, еще взяв старый учебник геометрии). Существуют альтернативные способы реализации нужных математических расчетов, но я не буду углубляться здесь в детали (ищите подсказки в пакете с примерами).

Даже если вы не хотите разбираться с математикой, просмотрите функцию `circle` в примере 11.11. По указанным координатам (X,Y) точ-

[1] Чтобы не выставлять программистов в невыгодном свете, следует отметить, что ко мне неоднократно обращались с просьбами прочитать лекции о программировании на языке Python для физиков, которые имели более богатую математическую практику, чем я, но многие из которых благополучно злоупотребляли общими блоками и операторами `GO TO` языка FORTRAN. Специализация в профессиональной деятельности может всех нас в чем-то превратить в новичков.

ки окружности, возвращаемым функцией `point`, она чертит линию из центра окружности в точку и маленький прямоугольник вокруг самой точки, что несколько напоминает стрелки и отметки аналоговых часов. Чтобы удалять нарисованные объекты перед каждым построением, используются теги холста.

Пример 11.11. PP4E\Gui\Clock\plotterGui.py

```
# рисует окружности на холсте

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():        # в 3.x // - деление с усечением
    circle(scaleVar.get(), (Width // 2), originX, originY, checkVar.get())

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 команд. строке?
```

```
originX = originY = Width // 2                  # то же, что и радиус
makewidgets()                       # в корневом окне Tk по умолчанию
mainloop()                          # в 3.x требуется // - деление с усечением
```

По умолчанию ширина круга составит 500 пикселей, если не определить иначе в командной строке. Получив число точек на окружности, этот сценарий размечает окружность по часовой стрелке при каждом нажатии кнопки Plot, вычерчивая прямые из центра к маленьким прямоугольникам на окружности. Переместите ползунок, чтобы задать другое число точек, и щелкните на флажке, чтобы рисование происходило достаточно медленно и можно было заметить очередность вычерчивания линий и точек (при этом сценарий вызывает `update` для обновления экрана после вычерчивания каждой линии). На рис. 11.19 приводится результат нанесения 120 точек при установке в командной строке ширины круга равной 400; если задать на окружности 60 или 12 точек, сходство с часовым циферблатом станет более заметным.

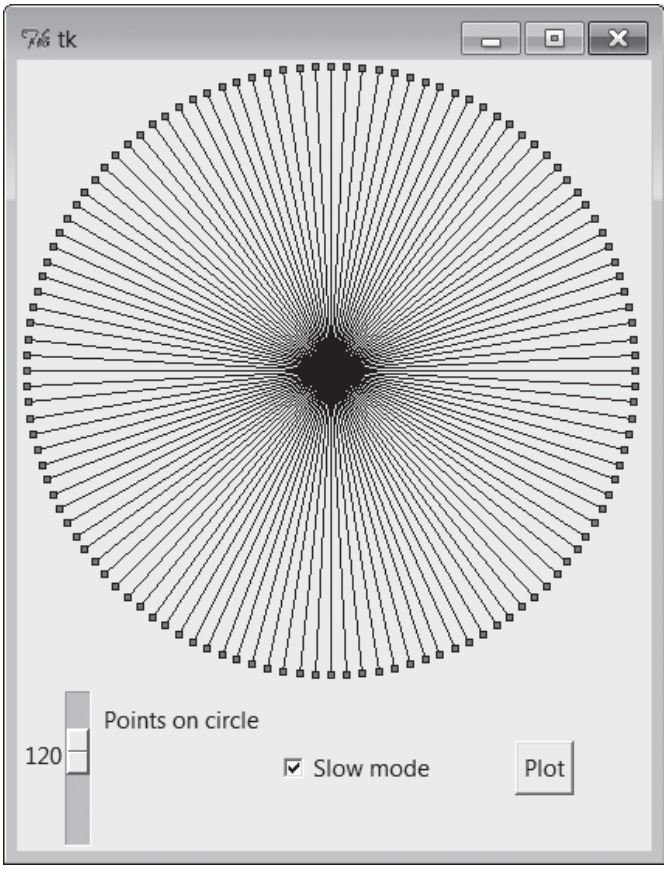


Рис. 11.19. *Сценарий plotterGui в действии*

Дополнительную помощь могут оказать ориентированные на текстовый, а не на графический вывод версии этого сценария, имеющиеся в дереве примеров, которые выводят координаты точек окружности в поток `stdout`, а не отображают эти точки в графическом интерфейсе. Смотрите сценарии *plotterText.py* в каталоге часов. Ниже показано, что он выводит для случаев 4 и 12 точек на окружности шириной 400 точек. Формат вывода прост:

```
номер_точки: угол = (координатаX, координатаY)
```

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)
----------
```

Инструменты Python для обработки чисел

Если вы сильны в математических расчетах настолько, чтобы разобраться в этом кратком уроке геометрии, вам, возможно, покажется интересным расширение NumPy для Python, предназначенное для поддержки численного программирования. В нем вы найдете такие объекты, как векторы, а также реализацию сложных математических операций, что превращает Python в инструмент решения научных задач, который эффективно реализует матричные операции и который сравним с MatLab. Расширение NumPy с успехом используется многими организациями, включая Ливерморскую и Лос-Аламосскую национальные лаборатории, – во многих случаях применение расширения NumPy позволяет писать на Python новые программы взамен устаревших программ на языке FORTRAN.

Расширение NumPy необходимо получать и устанавливать отдельно – смотрите ссылки на веб-сайте Python. В Интернете можно также найти родственные инструменты числовой обработки (например, SciPy), а также инструменты визуализации и трехмерной анимации (например, PyOpenGL, Blender, Maya, vtk и VPython). К моменту написания этих строк расширение NumPy (подобно многим числовым инструментам, опирающимся на его использование) официально доступно только для Python 2.X, однако версия, поддерживающая обе версии, 2.X и 3.X, уже находится в разработке[1]. Помимо модуля `math`, в языке Python имеется встроенная поддержка комплексных чисел для инженерных расчетов, в версии 2.4 появился тип десятичных чисел с фиксированной точностью, а в версии 2.6 и 3.0 была добавлена поддержка рациональных дробей. Подробности ищите в руководстве по стандартной библиотеке и в книгах, описывающих основы языка Python, таких как «Изучаем Python».

Чтобы понять, как эти точки отображаются на холст, нужно учесть, что ширина и высота окружности одинаковы и равны величине радиуса, умноженной на 2. Поскольку координаты холста tkinter (X,Y) начинаются с (0,0) в левом верхнем углу, центр окружности смещается в точку с координатами (ширина/2, ширина/2) – это будет точка, из которой вычерчиваются прямые. Например, в круге 400 на 400 центр холста будет в точке (200,200)[2]. Прямая в точку с углом 90 градусов (точка на правой стороне окружности) соединяет точку (200,200) и точку (400,200) – результат добавления к координатам центра координат точки (200,0), полученной для данного радиуса и угла. Линия вниз, к точке под углом 180 градусов, соединяет точки (200,200) и (200,400) после учета[3] координат вычисленной точки (0,–200).

Этот алгоритм построения точек, применяемый в `plotterGui`, а также несколько констант масштабирования лежат в основе отображения циферблата аналоговых часов в PyClock. Если вам все же кажется, что это слишком сложно, предлагаю сначала сосредоточиться на реализации

---

[1] В ноябре 2010 года вышла версия NumPu 1.5.1 для Python 3.1. – *Прим. перев.*

[2] Кроме этого следует учитывать, что, в отличие от координатных осей, используемых в планиметрии, на которых значения по оси Y возрастают снизу вверх (ось направлена вверх), на холсте ось Y направлена вниз. Это приводит к тому, что для вычисления на холсте позиции по Y вычисленная для точки координата Y **вычитается** из начальной координаты Y, а не прибавляется к ней, как в случае со значениями по оси X. – *Прим. ред.*

[3] После прибавления значения вычисленной координаты по X и вычитания значения по Y. – *Прим. ред.*

отображения *цифровых* часов. Аналоговые геометрические построения в действительности лишь являются расширением механизма отсчета времени, использующегося в обоих режимах отображения. В действительности в основе самих часов находится общий объект `Frame`, одинаковым образом посылающий *встроенным* объектам цифровых и аналоговых часов события изменения времени и размеров. Аналоговые часы – это прикрепленный виджет `Canvas`, умеющий рисовать окружности, а цифровые часы – просто прикрепленный фрейм `Frame` с метками, отображающими время.

Запуск PyClock

За исключением части, касающейся построения окружностей, программный код PyClock выглядит достаточно просто. Он рисует циферблат для отображения текущего времени и с помощью методов `after` вызывает 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 цифровых часов). На языке создания графических интерфейсов это означает, что аналоговое изображение выводится на холсте, перерисовывается при изменении размеров окна и изменяется по запросу на цифровой формат.

В PyClock используется также стандартный модуль Python `time`, с помощью которого сценарий получает и преобразует системную информацию о времени в представление, необходимое для часов. В двух словах, метод `onTimer` получает системное время вызовом функции `time.time`, встроенного средства, возвращающего число с плавающей точкой, выражающее количество секунд, прошедших с начала *эпохи,* – точки начала отсчета времени на вашем компьютере. Затем с помощью функции `time.localtime` это время преобразуется в кортеж, содержащий значения часов, минут и секунд. Дополнительные подробности можно найти в самом сценарии и руководстве по библиотеке Python.

Проверка системного времени 10 раз в секунду может показаться излишней, но она гарантирует перемещение секундной стрелки вовремя и без рывков и скачков (события `after` синхронизируются не очень точно). На компьютерах, которыми я пользуюсь, это не влечет существенного потребления мощности ЦП. В Linux и в Windows PyClock незначительно расходует ресурсы процессора – в основном при обновлении экрана в аналоговом режиме, но не в событиях `after`.[1]

[1] Например, сценарий PyDemos, представленный в предыдущей главе, запускает семь часов, выполняющихся в одном процессе, и во всех них обновление времени на моем (относительно медленном) нетбуке, работающем под управлением Windows 7, происходит плавно. Все вместе они потребляют единицы процентов мощности ЦП и нередко даже меньше, чем сама программа Диспетчер задач (Task Manager).

Чтобы минимизировать обновления экрана, PyClock перерисовывает только стрелки часов при переходе к следующей секунде – риски на циферблате перерисовываются только при начальном запуске и изменении размеров окна. На рис. 11.20 показан начальный циферблат PyClock в формате по умолчанию, который выводится при непосредственном запуске файла *clock.py*.

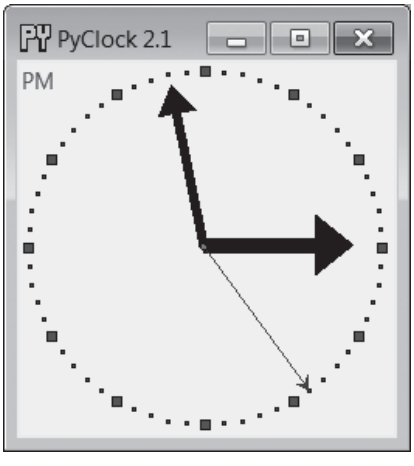


Рис. 11.20. *Аналоговые часы PyClock по умолчанию*

Линии, представляющие стрелки часов, имеют стрелку на одном конце, что определяется параметрами `arrow` и `arrowshape` объекта линии. Параметр `arrow` может принимать значение `first`, `last`, `none` или `both`; параметр `arrowshape` определяется как кортеж чисел, задающих длину стрелки на конце линии, общую длину линии и ее толщину.

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 левой кнопкой мыши на часах изменяет формат вывода на цифровой путем удаления виджета аналоговых часов и вывода цифрового интерфейса. В результате получается более простой интерфейс, изображенный на рис. 11.21.

Рис. 11.21. *Цифровые часы PyClock*

Такая цифровая форма может пригодиться, если вы хотите сберечь драгоценное место на экране и уменьшить использование ЦП (расходы на обновление изображения этих часов очень малы). Следующий щелчок

левой кнопкой на часах снова переводит их в аналоговый режим отображения. При запуске сценария конструируются оба отображения – аналоговое и цифровое, но в каждый отдельный момент прикреплено только одно из них.

Щелчок правой кнопкой мыши на часах в любом режиме отображения вызывает появление или исчезновение прикрепленной метки, показывающей текущую дату в простом текстовом формате. На рис. 11.22 показан аналоговый интерфейс PyClock с меткой даты и размещенной в центре фотографией (в таком виде часы запускаются из панели запуска PyLauncher).

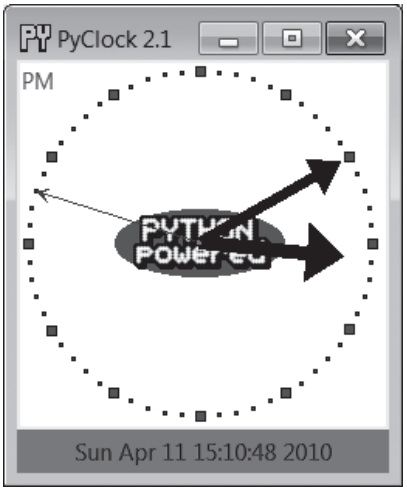


Рис. 11.22. *Улучшенный графический интерфейс PyClock с изображением*

Изображение в центре на рис. 11.22 добавлено путем передачи объекта с соответствующими настройками конструктору объекта PyClock. Почти все особенности этого изображения могут быть настроены через атрибуты объектов PyClock – цвет стрелок, цвет меток, центральное изображение и начальный размер.

Так как сценарий PyClock в аналоговом режиме сам отображает фигуры на холсте, ему необходимо также самостоятельно обрабатывать события *изменения размеров* окна: когда окно уменьшается или увеличивается, нужно перерисовывать циферблат часов в соответствии с новыми размерами окна. Чтобы реагировать на изменение размеров окна, сценарий регистрирует событие `<Configure>` с помощью метода `bind`; удивительно, но это событие не является событием менеджера окон, как событие для кнопки закрытия. Если растянуть окно PyClock, циферблат увеличится вместе с окном, – попробуйте растянуть окно часов, сжать или распахнуть его во весь экран на своем компьютере. Так как циферблат строится в квадратной системе координат, окно PyClock всегда

растягивается в равном отношении по вертикали и горизонтали – если растянуть окно только по горизонтали или только по вертикали, циферблат не изменится.

В третьем издании этой книги в часы был добавлен таймер обратного отсчета: нажатие клавиши s или m выводит простой диалог ввода числа секунд или минут, соответственно, через которое должен сработать таймер. По истечении отсчета таймера выводится всплывающее окно, как показано на рис. 11.23, заполняющее весь экран в Windows. Я иногда использую этот таймер на курсах, которые я веду, – для напоминания мне и моим студентам, когда подходит время двигаться дальше (эффект получается особенно потрясающий, когда изображение экрана компьютера проецируется во всю стену!).

Рис. 11.23. *Истек таймер PyClock*

Наконец, подобно PyEdit, часы PyClock можно запускать автономно или прикреплять и встраивать их в другие графические интерфейсы, где требуется вывести текущее время. При автономном запуске повторно используется модуль `windows` из предыдущей главы (пример 10.16) – чтобы установить значок и заголовок окна, а также добавить вывод диалога подтверждения перед выходом. Для упрощения запуска часов, выполненных в заданном стиле, существует вспомогательный модуль `clockStyles`, предоставляющий ряд объектов с настройками, которые можно импортировать, расширять в подклассах и передавать конструктору часов. На рис. 11.24 показано несколько часов разных размеров и стилей, подготовленных заранее, ведущих синхронный отсчет времени.

Запустите сценарий *clockstyles.py* (или щелкните на кнопке `PyClock` в программе PyDemos, которая делает то же самое), чтобы воссоздать эту сцену с часами на своем компьютере. Во всех этих часах 10 раз в секунду проверяется изменение системного времени с использованием со-

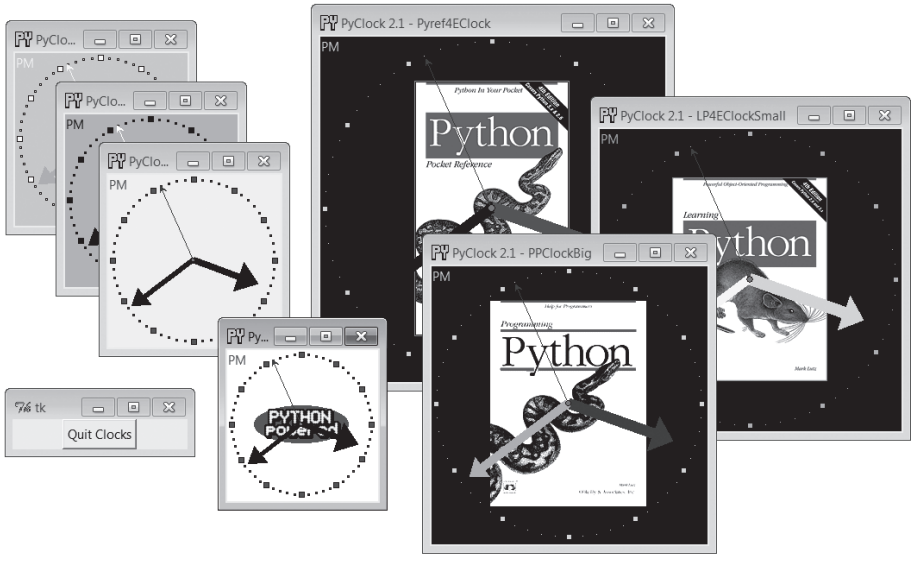


Рис. 11.24. *Несколько готовых стилей часов: clockstyles.py*

бытий `after`. При выполнении в виде окон верхнего уровня в одном и том же процессе все они получают событие от таймера из одного и того же цикла событий. При запуске в качестве независимых программ в каждой из них имеется собственный цикл событий. В том и другом случае их секундные стрелки дружно перемещаются раз в секунду.

Исходный программный код PyClock

Вся реализация PyClock находится в одном файле, за исключением предварительно подготовленных объектов с настройками стилей. Если посмотреть в конец примера 11.12, можно заметить, что объект часов можно создать, либо передав конструктору объект с настройками, либо определив параметры настройки в аргументах командной строки, как показано ниже (в этом случае сценарий просто сам создаст объект с настройками):

```
C:\...\PP4E\Gui\Clock> clock.py -bg gold -sh brown -size 300
```

Вообще говоря, для запуска часов этот файл можно выполнить непосредственно, с аргументами или без; импортировать его и создать объекты, используя объекты с настройками, чтобы часы выглядели более индивидуально; или импортировать и прикрепить его объекты к другим графическим интерфейсам. Например, PyGadgets из главы 10 запускает этот файл с параметрами командной строки, управляющими внешним видом часов.

Пример 11.12. PP4E\Gui\Clock\clock.py

```
"""
################################################################################
PyClock 2.1: часы с графическим интерфейсом на Python/tkinter.

В обоих режимах отображения, аналоговом и цифровом, могут выводить метку
с датой, графические изображения на циферблате, изменять размеры и так
далее. Могут запускаться автономно или встраиваться (прикрепляться) в другие
графические интерфейсы, где требуется вывести текущее время.

Новое в версии 2.0: клавиши s/m устанавливают таймер, отсчитывающий секунды/
минуты перед выводом всплывающего сообщения; значок окна.
Новое в версии 2.1: добавлена возможность выполнения под управлением Python 3.X
(2.X больше не поддерживается)
################################################################################
"""

from tkinter import *
from tkinter.simpledialog import askinteger
import math, time, sys

################################################################################
# Классы параметров настройки
################################################################################

class ClockConfig:
    # умолчания - переопределите в экземпляре или в подклассе
    size = 200                                      # ширина=высота
    bg, fg = 'beige', 'brown'                       # цвет циферблата, рисок
    hh, mh, sh, cog = 'black', 'navy', 'blue', 'red' # стрелок, центра
    picture = None                                  # файл картинки

class PhotoClockConfig(ClockConfig):
    # пример комплекта настроек
    size    = 320
    picture = '../gifs/ora-pp.gif'
    bg, hh, mh = 'white', 'blue', 'orange'

################################################################################
# Объект цифрового интерфейса
################################################################################

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) # TBD: при изменении размеров можно было бы
                                  # изменять размер шрифта
    def onUpdate(self, hour, mins, secs, ampm, cfg):
        mins = str(mins).zfill(2)                   # или '%02d' % x
        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 # здесь ничего перерисовывать не требуется

##############################################################################
# Объект аналогового интерфейса
##############################################################################

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:           # рисует овалы, 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    # 3.x деление //
            self.create_image(imgx+1, imgy+1, anchor=NW, image=self.image)
        originX = originY = radius = cfg.size // 2          # 3.x деление //
        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): # вызывается из
        if self.cog:                                 # обработчика событий
            self.delete(self.cog)                    # таймера, перерисовывает
            self.delete(self.hourHand)               # стрелки, центр
            self.delete(self.minsHand)
            self.delete(self.secsHand)
        originX = originY = radius = cfg.size // 2   # 3.x деление //
        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      # частота проверки системного времени

class Clock(Frame):
    def __init__(self, config=ClockConfig, parent=None):
        Frame.__init__(self, parent)
        self.cfg = config
        self.makeWidgets(parent) # дочерние виджеты компонуются методом pack,
        self.labelOn = 0         # но клиенты могут использовать pack или grid
        self.display = self.digitalDisplay
        self.lastSec = self.lastMin = -1
        self.countdownSeconds = 0
```

```
        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)
        parent.bind('<KeyPress-s>', self.onCountdownSec)
        parent.bind('<KeyPress-m>', self.onCountdownMin)

    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 += 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.countdownSeconds -= 1
            if self.countdownSeconds == 0:
                self.onCountdownExpire()            # таймер обратного отсчета
        self.after(1000 // ChecksPerSec, self.onTimer) # вызывать N раз в сек.
                                                        # 3.x // целочисленное
                                                        # деление с усечением
```

```
    def onCountdownSec(self, event):
        secs = askinteger('Countdown', 'Seconds?')
        if secs: self.countdownSeconds = secs

    def onCountdownMin(self, event):
        secs = askinteger('Countdown', 'Minutes')
        if secs: self.countdownSeconds = secs * 60

    def onCountdownExpire(self):
        # ВНИМАНИЕ: только один активный таймер,
        # текущее состояние таймера не отображается
        win = Toplevel()
        msg = Button(win, text='Timer Expired!', command=win.destroy)
        msg.config(font=('courier', 80, 'normal'), fg='white', bg='navy')
        msg.config(padx=10, pady=10)
        msg.pack(expand=YES, fill=BOTH)
        win.lift()                    # поднять над другими окнами
        if sys.platform[:3] == 'win': # в Windows - на полный экран
            win.state('zoomed')

################################################################################
# Автономные часы
################################################################################

appname = 'PyClock 2.1'

# использовать новые окна Tk, Toplevel со своими значками и так далее
from PP4E.Gui.Tools.windows import PopupWindow, MainWindow

class ClockPopup(PopupWindow):
    def __init__(self, config=ClockConfig, name=''):
        PopupWindow.__init__(self, appname, name)
        clock = Clock(config, self)
        clock.pack(expand=YES, fill=BOTH)

class ClockMain(MainWindow):
    def __init__(self, config=ClockConfig, name=''):
        MainWindow.__init__(self, appname, name)
        clock = Clock(config, self)
        clock.pack(expand=YES, fill=BOTH)

# для обратной совместимости: рамки окна устанавливаются вручную,
# передается родитель
class ClockWindow(Clock):
    def __init__(self, config=ClockConfig, parent=None, name=''):
        Clock.__init__(self, config, parent)
        self.pack(expand=YES, fill=BOTH)
        title = appname
        if name: title = appname + ' - ' + name
```

```
        self.master.title(title) # владелец=parent или окно по умолчанию
        self.master.protocol('WM_DELETE_WINDOW', self.quit)

##############################################################################
# Запуск программы
##############################################################################

if __name__ == '__main__':
    def getOptions(config, argv):
        for attr in dir(ClockConfig):       # заполнить объект с настройками
            try:                            # из арг. ком. строки "-attr val"
                ix = argv.index('-' + attr) # пропустит внутр. __x__
            except:
                continue
            else:
                if ix in range(1, len(argv)-1):
                    if type(getattr(ClockConfig, attr)) == int:
                        setattr(config, attr, int(argv[ix+1]))
                    else:
                        setattr(config, attr, argv[ix+1])

   #config = PhotoClockConfig()
    config = ClockConfig()
    if len(sys.argv) >= 2:
        getOptions(config, sys.argv)         # clock.py -size n -bg 'blue'...
   #myclock = ClockWindow(config, Tk())          # при автономном выполнении
   #myclock = ClockPopup(ClockConfig(), 'popup') # родителем является корневое
    myclock = ClockMain(config)                  # окно Tk
    myclock.mainloop()
```

И наконец, в примере 11.13 приводится модуль, выполняемый сценарием PyDemos, – в нем определяется несколько стилей часов и производится запуск одновременно семи экземпляров часов, прикрепляемых к новым окнам верхнего уровня для создания демонстрационного эффекта (хотя на практике обычно достаточно иметь на экране одни часы, даже мне!).

Пример 11.13. PP4E\Gui\Clock\clockStyles.py

```
# предопределенные стили часов

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'
    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 LP4EClock(GilliganClock):
    size = 700
    picture = gifdir + 'ora-lp4e.gif'
    bg = 'navy'

class LP4EClockSmall(LP4EClock):
    size, fg = 350, 'orange'

class Pyref4EClock(ClockConfig):
    size, picture = 400, gifdir + 'ora-pyref4e.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__':
    root = Tk()
    for configClass in [
        ClockConfig,
        PPClockBig,
        #PPClockSmall,
        LP4EClockSmall,
        #GilliganClock,
        Pyref4EClock,
        GreyClock,
        PinkClock,
        PythonPoweredClock
    ]:
        ClockPopup(configClass, configClass.__name__)
    Button(root, text='Quit Clocks', command=root.quit).pack()
    root.mainloop()
```

При запуске этот сценарий создает множество часов различного вида, как показано на рис. 11.24. Объекты конфигурации поддерживают большое число параметров. Судя по семи парам часов, отображаемых на экране, пришло время перейти к последнему примеру.

PyToe: виджет игры в крестики-нолики

И наконец, в завершение главы немного развлечемся. В нашем последнем примере, PyToe, на языке Python реализована программа игры в крестики-нолики с привлечением искусственного интеллекта. Большинству читателей, вероятно, знакома эта простая игра, поэтому я не стану останавливаться на ее описании. В двух словах: игроки поочередно ставят свои метки в клетках игрового поля, пытаясь занять целиком строку, колонку или диагональ. Победителем является тот, кому удалось сделать это первым.

В PyToe позиции на игровом поле помечаются щелчком мыши, а одним из игроков является программа на языке Python. Само игровое поле реализовано в виде простого графического интерфейса на основе tkinter. По умолчанию PyToe создает игровое поле размером 3 на 3 (стандартный вариант игры), но можно настроиться на игру произвольного размера *N* на *N*.

Когда приходит очередь компьютера сделать ход, с помощью алгоритмов искусственного интеллекта (ИИ) 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 победить на несколько ходов раньше, чем пользователь.

Запуск PyToe

Графический интерфейс PyToe реализован в виде фрейма с прикрепленными к нему метками и привязкой обработчиков событий щелчков мыши к этим меткам для перехвата ходов пользователя. Текст метки устанавливается равным метке игрока после каждого хода компьютера или пользователя. Здесь также повторно был использован класс `GuiMaker`, который мы создали ранее в предыдущей главе (пример 10.3), для создания простой полосы меню в верхней части окна (но без панели инструментов внизу, так как PyToe оставляет ее дескриптор пустым). По умолчанию пользователь ставит крестики («X»), а PyToe – нолики («O»). На рис. 11.25 показано игровое поле сценария PyToe, запущенного с помощью PyGadgets, и диалог с информацией о результатах игры; игра отображена на стадии, когда у сценария есть два хода, ведущие к победе.

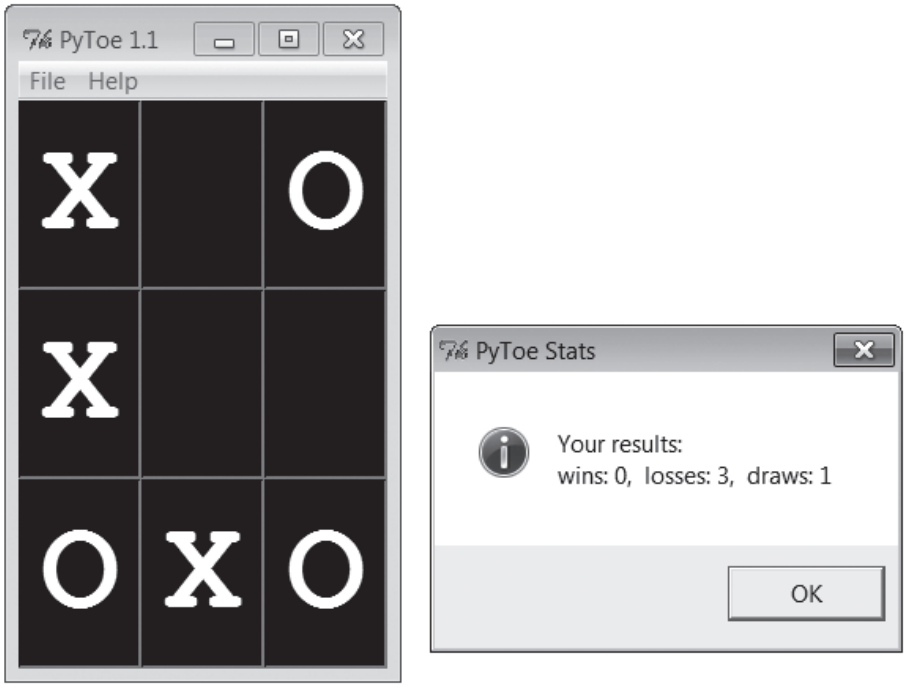


Рис. 11.25. *PyToe обдумывает путь к победе*

На рис. 11.26 изображен всплывающий диалог со справочной информацией о параметрах командной строки PyToe. Есть возможность определить цвет и размер для меток игрового поля, игрока, делающего первый ход, метку пользователя («X» или «O»), размер игрового поля (переопределяющий размер 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).

Используемая в PyToe технология ИИ интенсивно использует ЦП, и в зависимости от игровой ситуации компьютер тратит на определение следующего хода разное время, но скорость ответа компьютера зависит в основном от скорости компьютера. Задержка, связанная с выбором хода на игровом поле 3 на 3, составляет доли секунды для любой стратегии выбора хода «-mode».

На рис. 11.27 изображен альтернативный вариант настройки PyToe (сценарий PyToe был запущен непосредственно из командной строки без аргументов) в момент, когда программа только что выиграла у меня. Хотя по сценам игры, отобранным для этой книги, этого не скажешь, но при установке некоторых режимов выбора хода мне все же удается иногда выигрывать. На игровом поле большего размера и на более

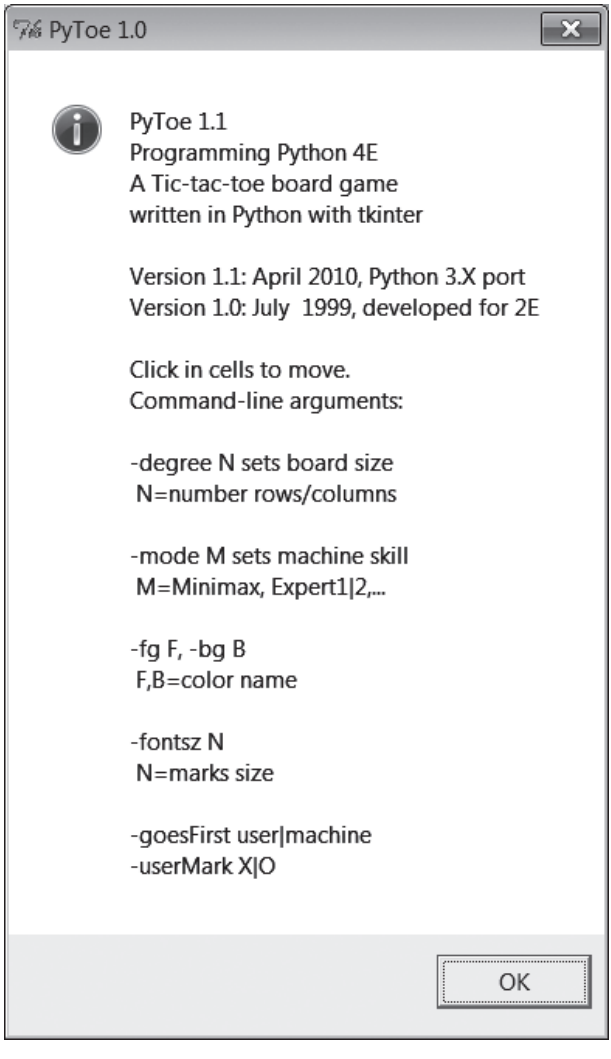


Рис. 11.26. *Диалог со справочной информацией о параметрах командной строки PyToe*

сложных уровнях алгоритм выбора хода, реализованный в PyToe, становится еще более эффективным.

Исходный программный код PyToe (внешний)

PyToe является крупной системой, для знакомства с которой предполагается наличие некоторой подготовки в области ИИ, но в отношении графического интерфейса, в сущности, не демонстрирует ничего ново-

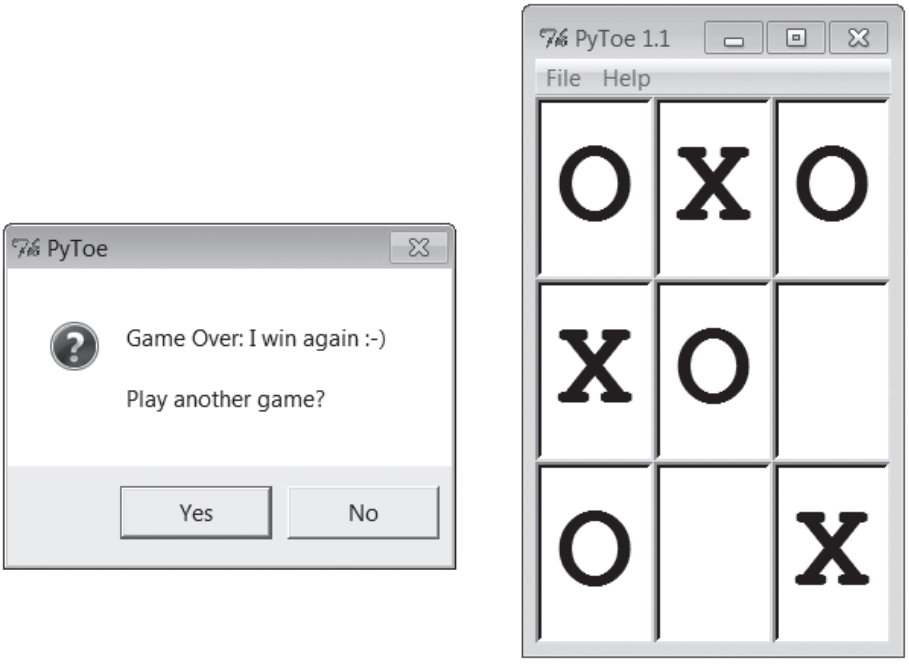


Рис. 11.27. *Альтернативный вариант настройки*

го. Кроме того, она была написана для выполнения под управлением Python 2.X более десяти лет тому назад, и хотя она и была перенесена на Python 3.X для этого издания, некоторые ее части было бы лучше реализовать заново. Отчасти по этой причине, но в основном из-за того, что я уже исчерпал объем страниц, отведенных на эту главу, я не привожу здесь исходный программный код, а отсылаю вас к пакету с примерами. За деталями реализации PyToe обращайтесь к следующим двум файлам из пакета примеров:

PP4E\Ai\TicTacToe\tictactoe.py

Сценарий оболочки верхнего уровня

PP4E\Ai\TicTacToe\tictactoe_lists.py

Основная реализация

Если вы решитесь заглянуть в эти сценарии, могу посоветовать обратить внимание на структуру данных, используемую для представления состояния игрового поля, которая составляет наибольшую сложность. Если вы разберетесь, каким образом моделируется игровое поле, то остальная часть реализации станет вполне понятна.

Например, в варианте, основанном на списках, для представления состояния игрового поля используется список списков, а также простой

словарь из виджетов полей ввода для графического интерфейса, индексируемый координатами игрового поля. Очистка игрового поля после игры заключается в простой очистке исходных структур данных, как показано в следующем фрагменте программного 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 игровое поле, и записать метку компьютера в нее и передать в графический интерфейс (атрибут `degree` хранит размер игрового поля):

```
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 конца 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 победа

def checkWin(self, mark, board=None):
    board = board or self.board
    for row in board:
        if row.count(mark) == self.degree: # проверка горизонтали
            return 1
    for col in range(self.degree):
        for row in board:                   # проверка вертикали
            if row[col] != mark:
                break
            else:
                return 1
    for row in range(self.degree):          # проверка первой диагонали
        col = row # row == col
```

```
            if board[row][col] != mark: break
        else:
            return 1
        for row in range(self.degree):          # проверка второй диагонали
            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 программный код, связанный с выбором хода, в основном просто проводит другие виды анализа структуры данных игрового поля или генерирует новые состояния для поиска в дереве ходов и контрходов.

В том же каталоге находятся родственные файлы, реализующие альтернативные схемы поиска и оценки ходов, различные представления игрового поля и так далее. За дополнительными сведениями об оценке ходов в игре и о поиске в целом обращайтесь к учебникам по ИИ. Это интересный материал, но слишком сложный, чтобы его можно было достаточным образом осветить в данной книге.

Что дальше

На этом завершается часть данной книги, посвященная графическим интерфейсам, но рассказ о графических интерфейсах на этом не заканчивается. Если вам необходимы дополнительные знания о графических интерфейсах, посмотрите примеры использования tkinter, которые будут встречаться дальше в книге и описаны в начале этой главы. PyMailGUI, PyCalc, а также внешние примеры PyForm и PyTree – все они представляют собой дополнительные примеры реализации графических интерфейсов. В следующей части книги мы также узнаем, как создавать интерфейсы пользователя, выполняемые в веб-броузерах, – совершенно другая идея, но еще один вариант конструирования простых интерфейсов.

Имейте в виду, что даже если ни один из рассмотренных в этой книге примеров графического интерфейса не похож на тот, который вам нужно запрограммировать, тем не менее вам уже были представлены все необходимые конструктивные элементы. Создание более крупного графического интерфейса для вашего приложения в действительности заключается в иерархическом расположении составляющих его виджетов, представленных в этой части книги.

Например, сложный интерфейс может быть реализован в виде совокупности радиокнопок, списков, ползунков, текстовых полей, меню и других виджетов, располагаемых во фреймах или в сетках для получения требуемого внешнего вида. Сложный графический интерфейс может быть дополнен всплывающими окнами верхнего уровня, а также независимо выполняемыми программами с графическим интерфейсом, связь с которыми поддерживается через механизмы взаимодействий между процессами (IPC), такими как каналы, сигналы и сокеты.

Кроме того, крупные компоненты графического интерфейса могут быть реализованы в виде классов Python, прикрепляемых или расширяемых всюду, где требуется аналогичный инструмент интерфейса, – важным примером таких компонентов может служить редактор PyEdit и его использование в PyView и PyMailGUI. Подойдите к делу творчески, и набор виджетов tkinter и Python обеспечит вам практически неограниченное число структур.

Помимо данной книги, посмотрите также документацию по библиотеке tkinter, представленную в главе 7, и книги, перечисленные на сайте Python *http://www.python.org* и в Интернете в целом. Наконец, если мне удалось вас увлечь библиотекой tkinter, еще раз хочу порекомендовать загрузить пакеты, о которых было сказано в главе 7, особенно Pmw, PIL, Tix и ttk (в настоящее время Tix и ttk входят в состав стандартной библиотеки), и поэкспериментировать с ними. Такие инструменты наращивают мощь арсенала tkinter, позволяя создавать более сложные графические интерфейсы за счет небольшого объема программного кода.

Алфавитный указатель

P

Q

R

T

Г

M

H

O

П

P

C