Роман Сузи

bhv®

www.bhv.ru
www.bhv.kiev.ua

Python



- Библиотеки Python Image Library и Numeric Python
- Работа с базами данных
- Среда разработки Web-приложений Zope

Наиболее полное руководство

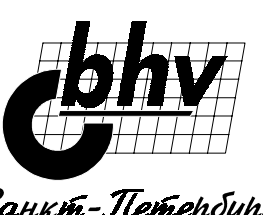
CD–ROM содержит инструментальные средства и коды примеров



В ПОДЛИННИКЕ®

Роман Сузи

Python



Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Описываются синтаксис и семантика типов данных, операций, конструкций и операторов языка Python; функции, специальные методы классов и исключения. Рассматриваются расширения языка: библиотека Numeric Python, позволяющая эффективно работать с массивами числовых данных, и библиотека Python Image Library, предназначенная для обработки растровых изображений. Обсуждаются работа с базами данных, возможности разработки графического интерфейса пользователя, интеграция программного продукта с модулями на языках SQL, Tcl/Tk, C, C++ и Java. Особое внимание уделено средствам Web-программирования. Прилагаемый компакт-диск содержит версии интерпретатора и значительную библиотеку программ и модулей расширения.

Для программистов и Web-разработчиков

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Руководитель проекта	*Анатолий Адаменко*
Зав. редакцией	*Анна Кузьмина*
Редактор	*Семен Штейнер*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Сузи Р. А.
Python. — СПб.: БХВ-Петербург, 2002. — 768 с.: ил.

ISBN 5-94157-097-X

© Р. А. Сузи, 2002
© Оформление, издательство "БХВ-Петербург", 2002

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 17.12.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 61,92.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9 линия, 12.

Содержание

Предисловие **1**

Введение **3**

1. Как построена эта книга 3
2. Используемые обозначения 5

Часть I. Основы программирования **9**

Глава 1. Интерактивный Python **11**

1.1. Запускаем Python 11
1.2. Диалог с Python 12
1.3. Python как калькулятор 14

Глава 2. Первые программы **19**

2.1. "Здравствуй, мир!" 19
2.2. Арифметика 20
2.3. Немного о функциях 21
2.4. Строки 23
2.5. Условия 26
2.6. Немного о логике 29
2.7. Циклы 31
2.8. Пишем комментарии 35
2.9. Учимся использовать списки 36
2.10. Работаем с текстом 40
2.11. Знакомимся с регулярными выражениями 41
2.12. Словари 47
2.13. Осваиваем форматированный вывод 50
2.14. Изучаем работу с файлами 54
2.15. Обработка ошибок 57

Глава 3. Функциональное программирование **60**

3.1. Чистые функции и их композиция 60
3.2. Обработка последовательностей 64
3.3. Рекурсия 67
3.4. Функции над функциями 71
3.5. Функции могут все 74

Глава 4. Строим модули **77**

4.1. Что такое модуль 77
4.2. Маленький проект 79

4.3. Интерфейсы модулей ... 83
4.4. Разрабатываем модули ... 86
4.5. Итоги нашего проекта ... 95

Глава 5. Объектный подход ... 96

5.1. Терминология ... 96
5.2. Наследование и агрегация ... 100
5.3. Используем готовые объекты ... 103
5.4. Строим свои классы ... 106
5.5. Модификация объектного проекта ... 112
5.6. Устойчивые объекты ... 114
5.7. Что дальше? ... 116

Глава 6. Графический интерфейс пользователя ... 117

6.1. Простой пример ... 118
6.2. Программа "Экзаменатор" ... 120
6.3. Игра "Сапер" ... 128

Часть II. Описание языка Python ... 135

Глава 7. Общие сведения о Python ... 137

7.1. История языка Python ... 137
7.2. О лицензиях ... 138
7.3. Области применения ... 139
7.4. Где взять Python и программное обеспечение ... 141
7.5. Установка Python ... 142
7.6. Где получить помощь ... 143

Глава 8. Описание языка ... 145

8.1. Структура программы ... 145
8.1.1. Физические и логические строки программы ... 146
8.1.2. Комментарии и пустые строки ... 147
8.1.3. Отступы и пробелы ... 149
8.2. Типы данных и объекты ... 151
8.2.1. Типы *None* и *Ellipsis* ... 154
8.2.2. Числа ... 154
8.2.3. Последовательности ... 157
Строки ... 158
Кортежи ... 162
Списки ... 163
8.2.4. Словари ... 167
8.2.5. Типы с поддержкой вызова ... 170
Функции ... 171
Методы ... 174
8.2.6. Классы и объекты ... 176
8.2.7. Модули ... 181

8.2.8. Тип 182
8.2.9. Файлы 183
8.2.10. Другие встроенные типы 186
Объекты-срезы 187
Кодовые объекты 188
Фрейм-объекты 189
Трассировочные объекты 189
8.3. Выражения 190
8.3.1. Атомы 195
Идентификаторы 195
Константы (литералы) 197
Задание кортежей, списков, словарей 201
Преобразование к строке 203
8.3.2. Первичные выражения 204
Ссылка на атрибут 204
Индексные выражения 205
Вызовы функций и других объектов 205
8.3.3. Операции 207
Арифметические операции 207
Операция форматирования 209
Логические операции 212
Сравнения 214
Побитовые операции 216
8.3.4. Лямбда-выражение 219
8.4. Специальные имена методов 220
8.4.1. Базовая настройка 220
Инициализация объекта: метод *__init__()* 220
Деструктор объекта: метод *__del__()* 221
Представление объекта в виде строки: методы *__repr__()* и *__str__()* 222
Сравнение объектов 223
Хэш-метод *__hash__()* 224
Проверка на истинность: метод *__nonzero__()* 225
8.4.2. Настройка доступа к атрибутам 226
Чтение атрибута: метод *__getattr__()* 226
Запись атрибута: метод *__setattr__()* 227
Удаление атрибута: метод *__delattr__()* 228
8.4.3. Эмуляция числовых типов 228
Методы для арифметических операций 228
Методы для побитовых операций 230
Методы для организации преобразований типов 230
Методы *__oct__()* и *__hex__()* 231
8.4.4. Эмуляция последовательностей и отображений 231
Метод *__len__()* 232
Метод *__getitem__()* 232
Метод *__setitem__()* 234
Метод *__delitem__()* 234
Методы *__getslice__()*, *__setslice__()* и *__delslice__()* 234

8.4.5. Поддержка вызова 235
8.5. Операторы 235
8.5.1. Операторы обработки данных 236
Оператор-выражение 236
Оператор присваивания 237
Оператор *print* 238
Оператор *del* 240
8.5.2. Управляющие операторы 241
Оператор *if* 241
Оператор *while* 243
Оператор *for* 244
Оператор *break* 246
Оператор *continue* 246
Оператор *return* 247
Оператор *pass* 247
8.5.3. Операторы обработки исключений 247
Оператор *try* 248
Оператор *raise* 251
Утверждения 252
8.5.4. Декларативные операторы 253
Оператор *global* 253
8.5.5. Операторы определения объектов 253
Оператор *def* 253
Оператор *class* 256
8.5.6. Операторы исполнения 260
Оператор *import* 260
Оператор *exec* 263
8.6. Встроенные функции и исключения 264
8.6.1. Встроенные функции 264
Функции преобразования типов 265
Числовые и символьные функции 268
Функции обработки данных 272
Функции определения свойств 277
Функции для работы с системными объектами 280
Функции компиляции и исполнения 283
Функции ввода/вывода 288
Функции для работы с атрибутами 290
Другие функции 291
8.6.2. Встроенные исключения 292
Базовые классы *Exception*, *SystemExit* и *StandardError* 293
ArithmeticError (базовый класс) 295
AssertionError 295
AttributeError 295
EnvironmentError (базовый класс) 295
FloatingPointError 296
ImportError 296

IndentationError....296
IndexError....296
IOError....297
KeyError....297
KeyboardInterrupt....297
LookupError (базовый класс)....298
MemoryError....298
NameError....298
NotImplementedError....298
OSError....299
OverflowError....299
RuntimeError....300
SyntaxError....300
SystemError....300
TabError....300
TypeError....301
UnboundLocalError....301
UnicodeError....301
ValueError....302
WindowsError....302
ZeroDivisionError....302
8.7. Пространства имен....302
8.8. Модули и пакеты....308
8.9. Оптимизация программ....312
8.10. О стиле программирования....314
8.11. Направления развития языка....318

Глава 9. Основные стандартные модули....322

9.1. Встроенная помощь: модуль *pydoc*....322
9.2. Сервисы периода исполнения....326
9.2.1. Модуль *sys*....326
9.2.2. Модуль *atexit*....329
9.2.3. Модуль *copy*....330
9.2.4. Модуль *gc*....333
9.2.5. Модуль *getpass*....335
9.2.6. Модуль *operator*....335
9.2.7. Модули *pickle* и *cPickle*....338
9.2.8. Модуль *pprint*....340
9.2.9. Модуль *repr*....341
9.2.10. Модуль *shelve*....343
9.2.11. Модуль *traceback*....344
9.2.12. Модуль *types*....346
9.2.13. Модуль *UserString*....348
9.2.14. Модули *UserList* и *UserDict*....349
9.2.15. Модули *__builtin__* и *__main__*....350

9.3. Обработка строк 350
9.3.1. Модуль *string* 350
9.3.2. Модуль *re* 355
9.3.3. Модули *StringIO* и *cStringIO* 364
9.3.4. Модуль *codecs* и работа с Unicode 366
9.3.5. Модуль *difflib* 371
9.3.6. Модуль *mmap* 374
9.4. Математические функции 376
9.4.1. Модуль *math* 376
9.4.2. Модуль *cmath* 378
9.4.3. Модули *random* и *whrandom* 379
9.5. Время и календарь 381
9.5.1. Модуль *time* 381
9.5.2. Модуль *calendar* 387
9.6. Массивы и структуры 388
9.6.1. Модуль *array* 389
9.6.2. Модуль *struct* 390
9.7. Взаимодействие с операционными системами (файлы, процессы) 394
9.7.1. Модуль *os* 394
Информация о системе 394
Параметры процесса 395
Создание и управление файловым объектом 397
Работа с файлами и каталогами 400
Создание и управление процессами 406
9.7.2. Модуль *os.path* 411
9.7.3. Модуль *dircache* 414
9.7.4. Модуль *errno* 414
9.7.5. Модуль *getopt* 416
9.7.6. Модуль *glob* 418
9.7.7. Модуль *popen2* 419
9.7.8. Модуль *shutil* 420
9.7.9. Модуль *select* 421
9.7.10. Модуль *signal* 423
9.7.11. Модуль *stat* 427
9.7.12. Модуль *tempfile* 430
9.7.13. Модуль *threading* 431
Класс *threading.Thread* 432
Класс *threading.Lock* 433
Класс *threading.RLock* 435
Семафоры: класс *threading.Semaphore* 435
Класс *threading.Event* 435
Класс *Condition* 437
9.7.14. Модуль *thread* 440
9.8. Простейшие базы данных. Архиваторы 443
9.8.1. Модуль *anydbm* 443
9.8.2. Модуль *whichdb* 444

9.8.3. Модуль *gzip* 444
9.8.4. Модуль *zlib* 446
9.8.5. Модуль *zipfile* 448
9.9. Модули для UNIX 451
9.9.1. Модули *pwd* и *grp* 451
9.9.2. Модуль *fcntl* 452
9.9.3. Модуль *resource* 455
9.9.4. Модуль *termios* 457
9.10. Редактирование в командной строке 459
9.10.1. Модуль *readline* 460
9.10.2. Модуль *rlcompleter* 462
9.11. Отладчик и профайлер 463
9.11.1. Модуль *pdb* 463
9.11.2. Модуль *profile* 465
9.12. Поддержка Internet. Протоколы 468
9.12.1. Модуль *cgi* 470
9.12.2. Модуль *urllib* 478
9.12.3. Модуль *urlparse* 481
9.12.4. Модуль *httplib* 483
9.12.5. Модуль *smtplib* 487
9.12.6. Модуль *poplib* 489
9.12.7. Модуль *telnetlib* 493
9.12.8. Модуль *socket* 495
9.13. Поддержка Internet. Форматы данных 502
9.13.1. Модуль *quopri* 502
9.13.2. Модуль *uu* 502
9.13.3. Модуль *base64* 503
9.13.4. Модуль *binhex* 504
9.13.5. Модуль *binascii* 504
9.13.6. Модуль *rfc822* 505
9.13.7. Модуль *mimetools* 508
9.13.8. Модуль *MimeWriter* 511
9.13.9. Модуль *multifile* 513
9.13.10. Модуль *mailbox* 515
9.14. Python и языки разметки (SGML, XML, HTML) 516
9.14.1. Модуль *sgmllib* 517
9.14.2. Модуль *htmllib* 521
9.14.3. Пакет *xml* 525
Пакет *xml.sax* 527
Модуль *xml.parsers.expat* 539
Пакет *xml.dom* 542
9.15. Элементы мультимедиа 550
9.15.1. Модуль *wave* 551
9.15.2. Модуль *colorsys* 553
9.15.3. Модули *sndhdr* и *imghdr* 554

9.16. Элементы криптографии 555
9.16.1. Модуль *md5* 555
9.16.2. Модуль *rotor* 557
9.16.3. Модуль *crypt* 558
9.17. Защищенная среда исполнения: модуль *rexec* 559
9.18. Python о себе 562
9.18.1. Модуль *keyword* 564
9.18.2. Модули *py_compile* и *compileall* 564
9.18.3. Модуль *dis* 565
9.19. Графический интерфейс. Python и Tk 566
9.19.1. Классы модуля *Tkinter* 567
9.19.2. Виджеты в общем 569
Создание виджетов 570
Базовые методы виджетов 570
Конфигурирование виджетов 576
Стилевые опции 577
Содержимое виджета 578
Задание шрифта. Модуль *tkFont* 579
Цвета 581
Описания событий 582
9.19.3. Различные виджеты 585
Окна верхнего уровня 585
Этикетка (виджет *Label*) 587
Кнопка (виджет *Button*) 588
Переключатель (*Radiobutton*) и флажок (*Checkbutton*) 590
Поле ввода (виджет *Entry*) 592
Меню 595
Текст (виджет *Text*) 596
Рисунок (виджет *Canvas*) 605
Менеджеры расположения 612
Переменные 616
9.19.4. Черепашья графика: модуль *turtle* 618
9.20. Модули для MS Windows 620
9.20.1. Модуль *msvcrt* 620
9.20.2. Модуль *_winreg* 621
9.20.3. Модуль *winsound* 625
9.21. Интернационализация и локализация 626
9.21.1. Модуль *locale* 626
9.21.2. Модуль *gettext* 631
9.22. Используем Distutils 636
9.22.1. Установка модулей 637
В среде Windows 637
В среде UNIX 638
9.22.2. Распространение модулей 640
Сценарий настройки 640
Файл конфигурации 640
Создание дистрибутивов 642
9.22.3. Список команд и опций 643

Часть III. Расширения Python 649

Глава 10. Python и C 651

Глава 11. Numeric Python 658

11.1. Модуль *Numeric* 658
11.2. Модуль *LinearAlgebra* 672

Глава 12. Обработка изображений: Python Image Library 675

12.1. Модуль *Image* 675
12.1.1. Примеры 676
12.1.2. Функции 677
12.1.3. Методы объектов класса *Image* 678
12.2. О других модулях 682

Глава 13. Связь с базами данных 686

13.1. Описание DB API 2.0 686
13.2. Модуль *MySQLdb* 691
13.3. Модуль для работы с PostgreSQL 695

Глава 14. Среда Zope 700

14.1. С точки зрения администратора 700
14.1.1. Установка Zope 701
14.1.2. Политика безопасности 702
14.1.3. Добавляем ресурс и пользователей 703
14.2. С точки зрения программиста 704
14.2.1. Zope-объекты 704
14.2.2. Знакомимся с языком DTML 705
14.2.3. Пишем сценарий 709
14.3. С точки зрения менеджера по содержанию 712

Глава 15. Jython — Python на Java 715

Часть IV. Приложения 723

Приложение 1. Переводы сообщений об ошибках 725

Приложение 2. Оболочка IDLE 728

Приложение 3. Описание компакт-диска 733

Список литературы 735

Предметный указатель 737

Предисловие

Еще три года назад мало кто в России знал о языке программирования с оригинальным названием Python, а уж, тем более, думал о его распространении в отрасли информационных технологий. Под языками программирования в то время было принято понимать языки системного программирования вроде C, Pascal, Basic, FORTRAN или Ada. И лишь Web-мастера и некоторые системные администраторы пользовались сценарным языком Perl, видя в нем мощное средство обработки текстовой информации.

Я познакомился с Python достаточно случайно: надо было найти простой язык программирования для школьного спецкурса. Выбор пал на Python, о котором кроме лестных эпитетов я ничего не знал... После короткого знакомства этот язык пленил меня своей элегантностью, краткостью и читаемостью кода.

По всей видимости, не существует гарантированных путей обучения программированию. Точнее, успех, скорее, зависит от учащегося, от того, насколько он настойчив. Умение программировать появляется внезапно, как озарение, при условии изучения предмета и некоторой практики. Я надеюсь, что первая часть этой книги как раз и приведет читателя к озарению, к вере в свои силы, послужит первой ступенькой в мир, в котором компьютеры управляются их пользователем с помощью программ, а не, наоборот, управляют им посредством нехитрых диалогов.

Конечно, этого мало. После того, как человек понял, что он в состоянии программировать, он способен освоить любую систему или язык программирования. На этом этапе очень важно привести его знания в стройную систему, которая позволила бы совершенствоваться без лишних зигзагов. На этом этапе нужно брать серьезные учебники и запастись терпением.

Став профессиональным разработчиком программного обеспечения, программистом, человек нуждается в литературе другого рода. Он уже знает основные понятия, связанные с программированием, и ему достаточно знать, как эти понятия отражаются в том или ином языке, где найти нужные библиотеки и какой интерфейс предоставляют классы, модули и пакеты. Иногда достаточно небольшого примера, чтобы разобраться в вопросе. Вторая часть книги поможет программистам сориентироваться в языке Python.

Менеджерам проектов, наоборот, не нужны детали: они должны представлять возможности технологии в крупных блоках. И здесь поможет третья часть книги, рассказывающая о применении Python.

Надеюсь, что Python понравится читателям, и они найдут ему достойное применение в своей работе.

Хотелось бы выразить признательность всем, кто помог, прямо или косвенно, работе над книгой, всему сообществу пользователей Python. И, конечно, разработчикам языка, модулей и расширений, которые ни на день не теряют контакта с пользователями: G. Van Rossum, F. Drake, B. Warsaw, J. Hylton, S. Montanaro, M. A. Lemburg, F. Lundh, T. Peters, A. Kuchling, M. von Loewis, N. Schemenauer, M. Zadka, E. S. Raymond, T. Heller, G. Ward, M. Hammond, P. Dubois, L. Garshol, U. Ogbuji, A. Fayolle, B. Herzog. Я также благодарен русской группе пользователей Python, в которой я консультировался по некоторым вопросам, связанным с Python и Zope, в особенности, О. Бройтману, К. Симонову, Д. Откидачу, Е. Двуреченскому, В. Хачатурову, В. Бормотову и др. Особая благодарность моей жене Наталье и дочери Алисе за всестороннюю поддержку и вдохновение.

Введение

Эта книга посвящена языку программирования Python. Наиболее точно и кратко о Python сказал его создатель — голландец Гвидо ван Россум (Guido van Rossum):

"Python — интерпретируемый, объектно-ориентированный высокоуровневый язык программирования с динамической семантикой. Его встроенные высокоуровневые структуры данных в сочетании с динамическим управлением типами и (поздним) динамическим связыванием делают язык очень привлекательным для быстрой разработки приложений (RAD, Rapid Application Development), а также в качестве сценарного или "склеивающего" языка для связи существующих программных компонентов. Простой, легкий для изучения синтаксис Python, в котором придается особое значение читаемости кода, сокращает затраты на сопровождение программ. Python поддерживает модули и пакеты, поощряя модульность и повторное использование кода. Интерпретатор Python и большая стандартная библиотека бесплатно доступны в виде исходных и бинарных кодов для всех основных платформ и могут свободно распространяться".

1. Как построена эта книга

Книга состоит из трех частей. Первая часть носит учебный характер — это введение в программирование с использованием языка Python. Вторая часть — справочная — описывает конструкции языка и наиболее важные библиотечные модули. Третья часть является обзором, знакомящим читателя с наиболее популярными расширениями и разработками, связанными с языком Python.

Рекомендуемый порядок чтения книги зависит от подготовленности читателя. Язык программирования Python задумывался своим создателем, в том числе, и как язык для "продвинутых" (advanced) пользователей, готовых стать программирующими пользователями. Для тех, кто никогда не писал программы, будут полезны "Основы программирования". Конечно, их нельзя считать исчерпывающим учебником по программированию, но благодаря многочисленным примерам они позволяют понять основные принципы и, возможно, лучше автоматизировать рутинную работу на компьютере.

Программисты и другие специалисты, желающие получить представление о Python, также могут начать с "Основ программирования" и продолжить свое знакомство с языком главами, посвященными расширениям Python. Напри-

мер, ученых может заинтересовать NumPy — расширение Python, позволяющее эффективно работать с массивами числовых данных *(глава 11)*. Web-мастерам могут приглянуться графическая библиотека Python Image Library (PIL) *(глава 12)*, работа с базами данных *(глава 13)*, с Zope *(глава 14)*. Python имеет широкое применение в обработке данных на XML — это обсуждается в *главе 9*. Разработчики программного обеспечения найдут полезной главу о "Python и Tk" *(глава 6)*, в которой показано применение Python в построении графического интерфейса пользователя.

Для программистов, непосредственно работающих на Python, будет наиболее интересна справочная часть книги, содержащая описания конструкций языка, объектов и модулей.

В книге очень много примеров. Примеры подобраны таким образом, чтобы дать как можно более полное представление о применении тех или иных объектов, функций, модулей. Так как Python имеет красивый и легко читаемый синтаксис, такие примеры часто говорят сами за себя. Опытному программисту достаточно одного взгляда на пример кода, использующего некоторую функцию, чтобы вспомнить особенности ее применения. Для новичка примеры имеют еще бóльшую ценность, т. к. являются основой для практических экспериментов с изучаемым языком.

Справочная часть этой книги не охватывает все модули и функции, поставляемые с Python (для этого есть оригинальное руководство). Она рассчитана на быстрое нахождение нужных функций (методов, модулей, операторов), исходя из стоящих задач. Например, если мы забыли, какая функция осуществляет поиск подстроки в строке, мы должны по оглавлению найти "Обработку строк" и прочитать список функций, отвечающий за обработку строк. Тем самым мы можем найти `string.index`, `string.find` и даже `string.replace` (если стоит задача замены найденных подстрок на другие). Кроме того, мы сразу увидим, что, начиная с Python 2.0, указанные функции являются методами объектов-строк. Таким образом, справочная часть отвечает на вопрос "Как в Python сделать то-то?" Ключевые слова выведены в предметный указатель. Например, нас интересует алгоритм сортировки последовательности данных. В предметном указателе мы находим слово "сортировка" и номера соответствующих страниц, открыв которые, мы сможем найти необходимые методы. Разумеется, если имя требуемого модуля нам известно, его легко найти в том же указателе.

При работе над книгой автор пользовался не только оригинальными руководствами (Language Reference, Library Reference и др.), но и заглядывал в исходные тексты описываемых модулей. Необходимо отметить, что оригинальная документация по Python выполнена очень качественно и поддерживается разработчиками в актуальном состоянии.

2. Используемые обозначения

Здесь будут рассмотрены некоторые условные обозначения, которые приняты в данной книге.

Даже просто пролистав книгу, можно заметить, что в ней поддерживается определенный стиль обозначений.

Для выделения в тексте фрагментов программ, команд, имен объектов, констант и тому подобного используется моноширинный шрифт. Символы строки заключаются в двойные кавычки (например, "\n"), тогда как имена операторов, функций, переменных употребляются в тексте книги без кавычек, например, print.

Термины, входящие в предметный указатель, выделены *курсивом.*

Клавиши, которые необходимо нажать для выполнения определенных действий, берутся в угловые скобки, например: <Ctrl>+<C> или <Ctrl>+ +<Break>. Другие обозначения оговариваются особо.

Операторы, состоящие из нескольких частей, обозначенных отдельными ключевыми словами, пишуться через дефис. Например, оператор обработки исключений обычно имеет две части: try и except, поэтому его можно называть "оператор try-except".

Наверное, не лишним будет объяснить и словосочетание "по умолчанию", столь часто употребляемое программистами. *Значением по умолчанию* называют значение, которое используется, если явно не задано какое-либо иное значение.

За очень редким исключением, примеры, представленные в данной книге, готовы к исполнению. Для этого их нужно поместить в файл и затем исполнить его интерпретатором Python. Именно с этой целью в каждом примере подключаются все необходимые модули, а не только рассматриваемый.

```
import urllib
print urllib.urlopen("http://www.python.org").read()[:400]
```

Результат выполнения:

```
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED.  DO NOT EDIT. -->
<!-- Wed Jan 10 15:34:39 2001 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:
Title: Python Language Website
-->
```

```
<HEAD>
<TITLE>Python Language Website</TITLE>
<META NAME="keywords" CONTENT="Python programming language object ori-
ented web free source">
```

В других примерах показан диалог с интерпретатором Python. В таких примерах команды и результаты их выполнения идут вперемежку. Команды предваряются характерным для Python приглашением `>>>` и вторичным приглашением `....` Полужирным выделено то, что нужно ввести.

```
>>> for i in range(20):
...   if i % 4 == 0:
...     print i,
...
0 4 8 12 16
>>> [ i for i in range(20) if i % 4 == 0 ]
[0, 4, 8, 12, 16]
```

Для описания синтаксиса функций, классов, методов и т. п. обычно используется запись следующего вида.

```
module.function(p1[, p2[, p3]]) -> res
```

где `module` — имя модуля, из которого берется функция, если она не является встроенной. Это может показаться излишним, т. к. обычно по контексту понятно, о каком модуле идет речь. Однако это позволяет быстрее ориентироваться в материале; `function` — название функции, класса, метода; `p1` — первый аргумент функции. Этот аргумент в нашем примере является обязательным и потому пишется вне скобок `[]`; `p2` — необязательный аргумент (заметьте использование скобок `[]`, которыми при описании синтаксиса принято обозначать необязательные части). При наличии необязательных аргументов обычно оговаривается и значение по умолчанию; `p3` — еще один необязательный аргумент; `res` — возвращаемый результат. Зачастую название этого элемента говорит о типе возвращаемых результатов: `i` — целое число, `0`, `1` — целые числа со значением ложь и истина, `s` — строка, `obj` — объект, о характере которого говорится в описании. Иногда при описании модуля одни и те же типы обозначаются одинаково. Например, при описании модуля `zlib` исходная строка обозначается `s`, а строка со сжатыми данными — `sc`.

Для упоминания сразу нескольких имен (например, `SOL_IP`, `SOL_SOCKET`, `SOL_TCP` и `SOL_UDP`) без их перечисления может быть сказано: `SOL_`-константы или даже константы `SOL*`. В таких случаях понятно, что речь идет о нескольких именах, т. к. звездочка не может быть частью имени.

Синтаксис операторов описывается с помощью нотации, похожей на расширенную *форму Бэкуса-Наура*. Ниже приведен пример использования такой нотации:

```
оп_def:        "def" имя "(" [форм_арг_ты] ")" ":" фрагмент
форм_арг_ты:   (поз_арг ",")* (спец_арг | поз_арг [","])
спец_арг:      "*" имя [, "**" имя] | "**" имя
поз_арг:       параметр ["=" выражение]
подсписок:     параметр ("," параметр)* [","]
параметр:      имя | "(" подсписок ")"
```

Слева записаны так называемые *нетерминальные символы* грамматики, а справа — смесь терминальных и нетерминальных символов. Самый первый нетерминальный символ (в нашем примере — `оп_example`) соответствует определяемой конструкции (здесь — оператор `def`). В результате замены нетерминальных символов терминальными и должна получаться правильная запись оператора. Терминальные символы обычно записываются в кавычках, как, например `"def"` или `")"`. Круглые скобки (не взятые в кавычки!) служат для группировки символов. Запись `(символ)*` означает, что `символ` может не встретиться вообще или встретиться один и более раз. Запись `(символ1|символ2)` означает, что на этом месте может находиться либо `символ1`, либо `символ2`. Если скобки отсутствуют, `|` действует для всего описания в данной строке (например, `спец_арг` может быть либо `"*" имя [, "**" имя]`, либо `"**" имя`. Квадратные скобки обозначают необязательность заключенных в них символов.

В оригинальной документации по Python и в файле Grammar, который поставляется вместе с исходными кодами Python, грамматика определяется более строго, чем в данной книге. Это связано с тем, что книга предназначена человеку, которому не нужно каждый раз расшифровывать, что такое `фрагмент` или `имя`, т. к. он может легко восстановить пробел, глядя на пример. В нашем случае таким примером может быть:

```
def EXAMPLE(x, y=None):
  if y is None:
    print "y не задан"
    y = 0
  return x+y
```

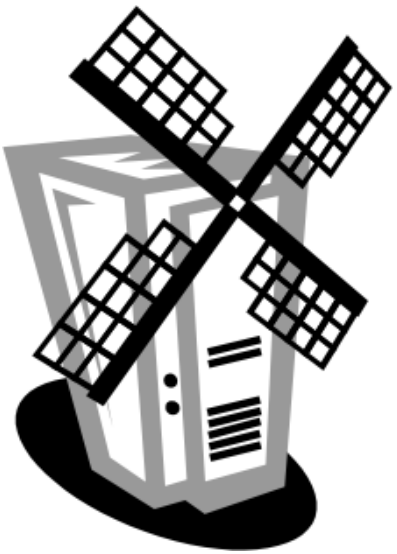
Здесь мы видим, что все терминальные символы (`def`, `(`, `)`, `=`, `...`) присутствуют в записи, а нетерминальные символы исчезли, наполнившись в соответствии со своим определением. Так, вместо `имен` появились `EXAMPLE`, `x` и `y`, а вместо `фрагмента` — целых четыре строчки с операторами.

При работе с примерами следует обязательно учитывать, что *регистр букв* (строчные или прописные) в подавляющем большинстве случаев является важным: `EXAMPLE`, `Example` и `example` — три различных имени. Пожалуй,

имеет смысл придерживаться одного и того же написания и в других ситуациях: при именовании файлов, использовании команд операционной системы, программировании на других языках программирования, даже если в них регистр букв не имеет значения. Поверьте, такая дисциплина позволит избежать лишних хлопот.

Примечание

В этой книге много примеров. Например, так, как этот текст, выделяются примечания, замечания, предупреждения и упражнения.



Часть I

Основы программирования

Основная задача этой части — начальное изучение программирования на примере языка Python. Здесь даны не только основные подходы к созданию программ, но и разобраны примеры решений различных задач обработки данных. Можно сказать, что в данной части мы имеем смесь теории и практики, позволяющую заинтересованному читателю не только понять программирование, но и успешно попрактиковаться.

Глава 1. Интерактивный Python

Глава 2. Первые программы

Глава 3. Функциональное программирование

Глава 4. Строим модули

Глава 5. Объектный подход

Глава 6. Графический интерфейс пользователя

Глава 1



Интерактивный Python

В этой главе мы познакомимся с Python — новым интересным языком программирования. Он задумывался для тех, кто хочет с легкостью и изяществом решать даже трудные задачи, и подходит для тех, кто хочет научиться программировать.

1.1. Запускаем Python

Для диалога с Python нам необходимо прежде всего запустить интерпретатор Python. Упрощенно, *интерпретатор* — это такая программа, которая последовательно, одну за другой, читает команды языка программирования (они называются операторами), и переводит их в машинные команды, которые исполняет процессор компьютера.

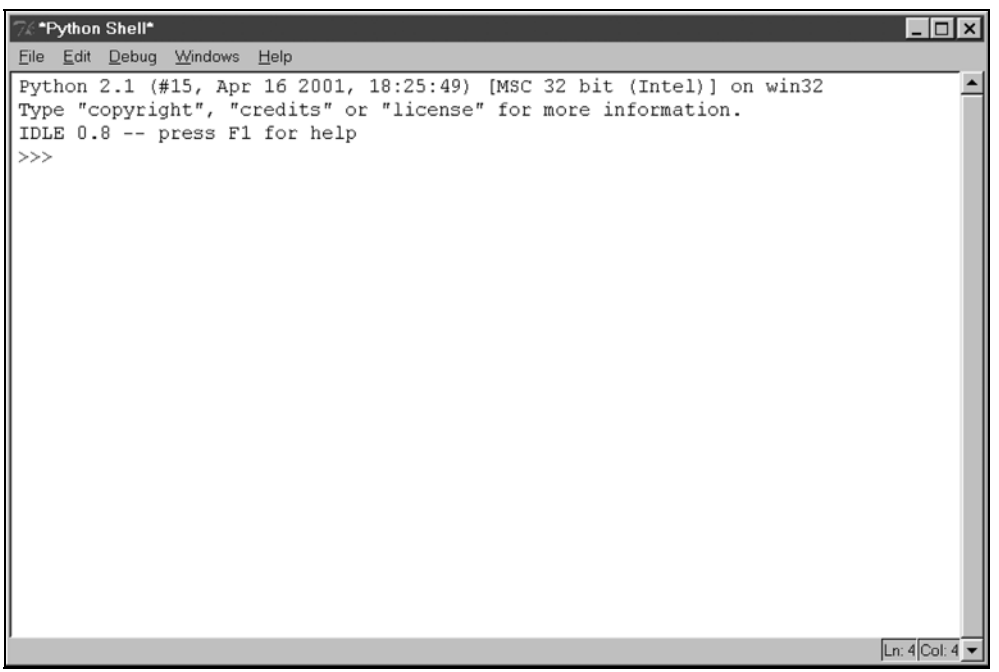


Рис. 1.1. Первый запуск IDLE

В системе Windows интерпретатор обычно запускается вместе с графической оболочкой IDLE (рис. 1.1) из **Программы | Python | IDLE (Python GUI)**, но можно запустить только интерпретатор, выбрав команду **Программы | Python | Python (Command line)**. И то, и другое даст нам возможность вводить команды, которые Python будет исполнять.

В системе UNIX и Linux также можно запустить Python просто командой `python`, а можно запустить IDLE (если вы любите работать в X Window) — командой `idle`. (В UNIX режим командной строки гораздо богаче возможностями редактирования, чем в Windows).

Сразу после запуска Python нас приветствует, например, так:

```
Python 2.1 (#1, Apr 21 2001, 13:46:58)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.
>>>
```

Три знака `">"` — *приглашение ко вводу*. Мы можем вводить наши команды.

1.2. Диалог с Python

Наш Python готов к диалогу, о чем говорит `>>>`. Для начала спросим его о погоде на сегодня:

```
>>> Какая сегодня погода?
  File "<stdin>", line 1
    Какая сегодня погода?
                ^
SyntaxError: invalid syntax
>>>
```

К сожалению, уважаемый Python не понимает русского языка, и нам придется учить его язык. Но что же он нам сказал, указав на нашу ошибку? `SyntaxError` — синтаксическая ошибка, а более конкретно `invalid syntax` — неверный синтаксис. Высказавшись, Python снова предложил продолжить общение.

```
>>> 2+2
4
```

Вот это ему понятно. Пробуем дальше (`print` означает "печатай"):

```
>>> print "Сегодня хороший день!"
Сегодня хороший день!
>>> print "Сегодня хороший день!", 1, "мая!"
Сегодня хороший день! 1 мая!
```

```
>>> print "Завтра будет", 1+1, "мая!"
Завтра будет 2 мая!
>>> print "Послезавтра будет", 1+ "мая!"
Послезавтра будет
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand types for +
>>>
```

Кажется, мы увлеклись и забыли, что компьютер — вещь точная, и требует четкости при работе с ним. На этот раз произошла ошибка типа `TypeError`, потому что нельзя складывать числа и строки, как это случайно произошло в диалоге выше. Ничего страшного, мы исправим ошибку и продолжим общение:

```
>>> print "Послезавтра будет", 1+2, "мая!"
Послезавтра будет 3 мая!
>>>
```

Кстати, строки с текстом тоже можно "складывать":

```
>>> print "Послезавтра" + "будет" + "май!"
Послезавтрабудетмай!
>>> print "Послезавтра " + "будет " + "май!"
Послезавтра будет май!
```

А вот арифметические операции в кавычках исполняться, естественно, не будут:

```
>>> print "2+2"
2+2
```

Python все равно, использованы ли кавычки или апострофы. Кроме того, для записи строк текста можно применять утроенные кавычки или апострофы:

```
>>> print """Текст можно писать и в таких кавычках, """ 'и в таких.'
Текст можно писать и в таких кавычках, и в таких.
>>> print "Но не в таких:", `2+2`
Но не в таких: 4
```

Надеемся, что читатель неплохо провел время в легкой беседе с интерпретатором Python и готов к более серьезному общению в следующем разделе.

Упражнение

Сложите яблоки с апельсинами.

1.3. Python как калькулятор

В этом разделе мы научимся использовать Python в качестве калькулятора. Запустите Python (*см. разд. 1.1*, если вы забыли, как это делается) и за дело.

Python умеет производить простейшие арифметические операции: сложение, вычитание, умножение, деление и даже получение остатка от деления. Причем делать он это может как с целыми числами, так и с десятичными дробями, а также с комплексными числами. В записываемых числовых примерах (их грамотно называть *выражениями*) можно расставлять скобки так, как нужно для правильных вычислений. В отличие от многих калькуляторов, Python знает, что умножение, деление и получение остатка от деления имеет бóльший приоритет, чем сложение или вычитание. Давайте посмотрим, как Python справится со следующими выражениями:

```
>>> 2+3*4
14
>>> 2*3+4
10
>>> (2+3)*4
20
>>> 10/2
5
>>> 10/3
3
>>> 12.48/2
6.24
>>> 13%5
3
```

В первых примерах все ясно. Операция умножения обычно обозначается в языках программирования как `"*"`, деление — как `"/"`, а операция получения остатка от деления — как `"%"`. Вы, наверное, заметили, что десятичные дроби пишутся через точку (`12.48`), несмотря на то, что в России принято писать через запятую. Запись дробей с точкой традиционна для многих языков и систем программирования. Кроме того, результат `10/3` равен `3`. Дело в том, что и `10`, и `3` — целые числа, и потому Python осуществляет деление нацело. Для того чтобы разделить числа "обычным" образом, нужно записать хотя бы одно из них как десятичную дробь — с точкой:

```
>>> 10.0/3
3.33333333333
>>> 11./3
```

```
3.66666666667
>>> 10/3.
3.33333333333
```

Теперь можно решать примеры любой сложности:

```
>>> (12.3 + 12)/(12 - (78.3+5*6))
-0.252336448598
>>> _ + 5
4.7476635514
```

Одиночное подчеркивание "_" является специальным именем для обозначения результата предыдущего вычисления.

Упражнение

Составьте выражение на Python, которое вычисляет стоимость покупки яблок, мандаринов и бананов, если известны их вес и цена за килограмм.

Следующий пример показывает, что некоторые операции над большими числами приводят к переполнениям, о которых Python тут же заявляет:

```
>>> 1000000000*1000000000
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OverflowError: integer multiplication
```

Это было сказано для того, чтобы, пользуясь имеющейся в Python операцией возведения в степень (обозначается **), вы не удивлялись возникающим переполнениям:

```
print "1 килобайт =", 2**10, "байт"
1 килобайт = 1024 байт
print "1 мегабайт =", 1024 ** 2, "байт"
1 мегабайт = 1048576 байт
print "1 гигабайт =", 1024 ** 3, "байт"
1 гигабайт = 1073741824 байт
>>> 2**100
Traceback (innermost last):
  File "<stdin>", line 1, in ?
OverflowError: integer pow()
```

Упражнение

В какую наибольшую степень можно возвести число 2 без переполнения?

Python может стать "инженерным" калькулятором, если перед вычислениями подать специальную команду, загружающую в память модуль с различными математическими функциями *(см. разд. 9.4.1)*. Ниже мы будем использовать `sqrt` (квадратный корень), `sin` (синус), `cos` (косинус), `atan` (арктангенс).

```
>>> from math import *
>>> sqrt(16)
4.0
>>> sqrt(18.49)
4.3
>>> sin(3)
0.14112000806
>>> sin(1.22)**2 + cos(1.22)**2
1.0
```

Упражнение

Поработав с функцией `fabs()`, попытайтесь установить, что она делает.

Работая с обычными калькуляторами, бывает очень обидно, что с таким трудом вычисленные значения нужно стирать перед выполнением других вычислений. В Python результатам вычислений можно давать *имена*:

```
>>> from math import *
>>> x = sin(0.5) + atan(2.3)
>>> y = 7
>>> x*y
11.480661674
```

В этом примере Python, вычислив `sin(0.5) + atan(2.3)`, дает полученному результату имя `x`. Аналогично происходит и с `y`, хотя там уже никаких сложных вычислений нет. Говорят, что (переменной) `y` присваивается значение `7`. А дальше мы просто используем имена переменных в вычисляемых выражениях наравне с числами.

Кстати, все тот же модуль `math` содержит имена, соответствующие приближенным значениям известных констант π и e:

```
>>> from math import *
>>> print "pi =", pi
pi = 3.14159265359
>>> print "e =", e
e = 2.71828182846
```

Так что площадь круга радиуса `r` можно вычислить так:

```
>>> from math import *
>>> r = 3
>>> print "S =", pi * r**2
S = 28.2743338823
```

Упражнение

Воспользуйтесь интерпретатором Python для вычисления по какой-либо известной вам формуле.

За исключением примера с переполнением, в наших вычислениях участвовали сравнительно небольшие числа. Оказывается, для представления очень больших или очень близких нулю чисел существует свой вид записи — *запись в экспоненциальном формате*. В такой записи записываются лишь действительные числа (числа "с точкой"):

```
>>> 0.0001
0.0001
>>> 0.00001
1e-05
>>> 10.0**-6
1e-06
>>> 100000000000.
100000000000.
>>> 1000000000000.
1e+12
>>> 10.0**13
1e+13
>>> 3.14e5
314000.0
>>> 3.14e-5
3.14e-05
>>> 3.14e-3
0.00314
```

Секрет записи раскрывается просто: число получается в результате умножения *мантиссы* (стоит перед `"e"`) на 10 в степени, указанной после `"e"`. Указанное после `"e"` число называется *порядком*.

Упражнение

Как в Python запишется миллион миллиардов?

Если же нас интересуют большие целые числа, мы можем использовать целые числа с произвольной точностью. В Python они (скромно) названы *длинными целыми*. Запись длинного целого отличается от записи обычного целого только добавлением "L" на конце:

```
>>> print "1 терабайт =", 1024L ** 4, "байт"
1 терабайт = 1099511627776L байт
>>> print "1 петабайт =", 1024L ** 5, "байт"
1 петабайт = 1125899906842624L байт
```

Числа с произвольной точностью ограничены лишь размером памяти, доступной интерпретатору Python, и степенью вашего терпения при ожидании результатов вычислений.

Упражнение

Вычислите с помощью целых чисел произвольной точности число 2 в десятитысячной степени. А в стотысячной?

Кстати, выполняющиеся вычисления можно прервать с помощью комбинации клавиш <Ctrl>+<C> (Linux) или <Ctrl>+<Break> (Windows). Прервемся и мы, чтобы в следующей главе заняться настоящими программами.

Глава 2



Первые программы

Чем больше работаешь в диалоговом режиме с интерпретатором, тем больше хочется составлять полноценные *программы* — предписания компьютеру на понятном ему языке — и записывать их в файл на диске. В этой главе читатель постигнет азы программирования с помощью языка Python.

Под *программированием* компьютера мы будем понимать искусство заставить компьютер работать по заранее составленной нами программе. Просто работу неподготовленного пользователя, например, в текстовом процессоре, программированием, конечно, не назовешь. Но когда тот же пользователь научился создавать хотя бы небольшие программы — это уже подходит под наше определение.

Здесь мы считаем, что Python установлен в системе и читатель умеет пользоваться простейшим текстовым редактором для ввода и сохранения текстовых файлов.

2.1. "Здравствуй, мир!"

По сложившейся традиции, первая программа, которую рассматривают при изучении какого-либо языка программирования, должна выводить слова "Hello, World!" ("Здравствуй, мир!"). С помощью текстового редактора (Блокнот, Просто Текст, mcedit и т. п.) нужно записать в файл hello.py следующий текст:

Листинг 2.1. Файл hello.py. Первая программа

```
print "Hello, World!"
print "Здравствуй, мир!"
```

и запустить на выполнение из командной строки:

```
python hello.py
```

"Да это же... Basic!" — воскликнет посвященный читатель. И правда, написанный фрагмент может исполнить и интерпретатор Basic (понимающий отсутствие номеров строк). На Python можно записать и так, заключив текст в утроенные кавычки:

Листинг 2.2. Файл hello2.py. Первая программа (второй вариант)

```
print """Hello, World!
Здравствуй, мир!"""
```

Программа по-прежнему выводит:

```
Hello, World!
Здравствуй, мир!
```

2.2. Арифметика

Проверим, умеет ли Python считать по программе. Подсчитаем площадь круга по радиусу:

```
import math
print "Введите радиус круга r=",
r = input()
S = math.pi * r ** 2
print "Площадь круга радиуса", r, "равняется", S
```

Рассмотрим программу подробнее. Первая строка дает Python расширенные познания в математике. Теперь он знает константу π, необходимую для вычислений. Эта константа берется из модуля `math`, содержащего целый набор математических функций. Мы просто импортируем (присоединяем к нашей программе) модуль и используем π из него. Во второй строке программа просит пользователя ввести значение `r`, и в третьей функция `input()` вводит его. Собственно вычисления происходят в четвертой строке. Здесь π умножается на квадрат радиуса. В Python операция возведения в степень обозначается `'**'`, как в FORTRAN. Результат называется `S`. Пятая строка выводит сообщение, состоящее из поясняющих строк и выводимых величин через запятую.

Замечание

Здесь мы видим другой способ обращения к функциям из модуля `math`, отличающийся от того, который мы знаем по предыдущему разделу.

Оказывается, это еще не все. Примененная нами функция `input()` умеет больше! Например, мог бы иметь место следующий диалог:

```
Введите радиус круга r=3*25+4
Площадь круга радиуса 79 равняется 19606.66319
```

Мы видим, что функция ввода `input()` имеет встроенный "калькулятор".

Упражнение

Напишите программу, которая по заданному радиусу и высоте вычислит объем цилиндра. Указание: объем цилиндра равен произведению площади основания на высоту.

2.3. Немного о функциях

Мы уже написали несколько программ, и все-таки создается такое впечатление, что они какие-то "одноразовые": если нужно подсчитать несколько площадей приходится копировать текст формулы несколько раз. К счастью, созданный однажды код можно исполнять снова и снова и без копирования, если поместить его в *подпрограмму*. Подпрограмма решает подзадачу — часть той задачи, которая возложена на программу.

В языке Python подпрограммы называются функциями и обычно определяются оператором `def`. Внутри оператора `def` делается отступ, который и обозначает границы текста определения функции. Оператор `return` возвращает вычисленное значение. Схематично определение функции можно записать так:

```
def имя(аргументы):
  операторы определения функции
  return результат
```

Здесь `def` происходит от слова *to define* (определить), а *return* по-английски означает "возвратить". Операторов `return` может быть не один, а может не быть ни одного. Правда, в последнем случае функция будет возвращать специальное значение `None`.

Перепишем программу подсчета площади круга с применением функции (то, что записано в строках после `#`, является комментарием):

Листинг 2.3. Функция по вычислению площади круга

```
import math
# Определяем функцию:
def Square(radius):       # определим Square с аргументом radius
  return math.pi * radius**2  # возвращаем "пи эр квадрат"

# Проверяем работу функции:
print "Площадь круга радиуса", 2, "равняется", Square(2)
print "Площадь круга радиуса", 4, "равняется", Square(4)
print "Отношение площадей:", Square(4)/Square(2)
```

Результат выполнения:

```
Площадь круга радиуса 2 равняется 12.5663706144
Площадь круга радиуса 4 равняется 50.2654824574
Отношение площадей: 4.0
```

Теперь мы убрали детали вычисления площади круга в функцию и можем проявить творчество. Например, подсчитать отношение площадей двух кругов.

Рассмотрим еще один пример, когда функция имеет два аргумента:

Листинг 2.4. Функция, вычисляющая площадь прямоугольника по двум сторонам

```
def Square1(a, b):        # определим Square1 с аргументами a и b
  return a * b

print "Площадь прямоугольника", 2, "x", 3, ":", Square1(2, 3)
print "Площадь прямоугольника", 8, "x", 3, ":", Square1(8, 3)
```

Результат выполнения:

```
Площадь прямоугольника 2 x 3 : 6
Площадь прямоугольника 8 x 3 : 24
```

Упражнение

Перепишите программу для вычисления объема цилиндра из упражнения предыдущего раздела с использованием функций. Подсказка: объем цилиндра вычисляется как произведение площади его основания на высоту.

В Python имеется большое число готовых к использованию функций. Часть из них доступна сразу, для доступа к другим нужно подключать дополнительные модули.

Использованный нами оператор `import` имеет две разновидности.

- `import модуль1, модуль2, ...`
- `from модуль import имя1, имя2, ...`

В первом случае мы подключаем к нашей программе заданные в операторе модули, а во втором мы берем из (по-англ. — *from*) одного модуля несколько определенных там имен функций или переменных. В следующих разделах мы будем частенько использовать оператор `import`, а в *гл. 4* научимся делать свои собственные модули.

2.4. Строки

До сих пор мы работали в основном с числами, вычисляли по формулам. Однако с помощью компьютера можно не только вычислять. Python позволяет обрабатывать текстовые данные, и в этом разделе мы изучим некоторые функции для работы со строками символов.

Примечание

Обычно тексты хранятся в файлах на диске, но работу с файлами мы изучим только в *разд. 2.14*, а пока будем записывать текст прямо в программе.

Строки состоят из символов. Каждый символ обозначает некоторую букву, цифру, знак препинания или другой символ. Кроме того, некоторые символы имеют специальное значение и никак не отображаются. К таким символам относится символ конца строки. В строке на Python его можно задать комбинацией двух символов `"\"` и `"n"`. Каждый символ имеет числовой код от 0 до 255. Символы с кодами от 0 до 127 стандартизированы и называются набором символов *ASCII* (American Standard Code for Information Interchange, американский стандартный код для обмена информацией). Остальные коды могут использоваться по-разному. Обычно они используются для символов национальных алфавитов. Следующий пример показывает, как можно узнать код символа и, наоборот, воспроизвести символ по коду:

```
print ord("D")
print chr(68)
```

Получаем:

```
68
D
```

Следующий пример показывает простейшие операции над строками — соединение (более научно — конкатенация) и размножение:

```
print "ковер" + "-" + "самолет"
print "~" * 13
```

Получаем:

```
ковер-самолет
~~~~~~~~~~~~~
```

Сначала мы соединили три строки в одну, а затем размножили символ `"~"` тринадцать раз.

Строковым значениям, как и числам, можно давать имена:

```
s1 = "кад"
s2 = "абра"
```

```
s = s2 + s1 + s2
print s
```

Результат выполнения:

```
абракадабра
```

В следующих примерах показаны различные функции и операции со строками.

Длину строки можно узнать, применив встроенную функцию `len()`:

```
>>> s = "Это текст."
>>> print "Длина строки:", len(s)
Длина строки: 10
```

А в следующем примере мы вырезаем из строки один или несколько символов:

Листинг 2.5. Примеры вырезки символов из строки

```
s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print "Вырезаем символы с 3-го по 5-й:", s[3:6]
print "Вырезаем символы с 7-го до конца строки:", s[7:]
print "Вырезаем 6 символов от начала строки:", s[:6]
print "Берем третий символ с конца:", s[-3]
```

Результат выполнения:

```
Вырезаем символы с 3-го по 5-й: DEF
Вырезаем символы с 7-го до конца строки: HIJKLMNOPQRSTUVWXYZ
Вырезаем 6 символов от начала строки: ABCDEF
Берем третий символ с конца: X
```

К этой программе необходимы некоторые пояснения. Python считает не символы, а промежутки между ними. Началу строки соответствует `0`, поэтому первый символ можно вырезать с помощью `[0:1]`. Для простоты в случае одного символа можно просто написать `[0]`. Поэтому можно считать, что символы строки нумеруются с нуля. Символы можно считать и с конца строки. Промежуток между предпоследним и последним символом имеет номер `-1`, поэтому для взятия предпоследнего символа можно написать `[-2:-1]` или просто `[-2]`. Последние пять символов можно взять с помощью `[-5:]`. Неуказанная правая граница означает "до конца строки". Точно так же неуказанная левая граница означает "от начала строки".

На рис. 2.1 мы видим строку `"Example"` с пронумерованными промежутками между символами. Python умеет считать промежутки между символами как от начала, так и с конца строки. В последнем случае нумерация идет со знаком минус. Например, `"Example"[-6:-5]` даст нам строку `"x"`, а `"Example"[1:4]` — строку `"xam"`.

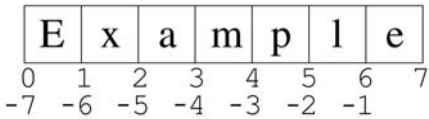


Рис. 2.1. Объяснение нумерации промежутков между символами строки

В отличие от языка Basic, строковые значения в языке Python изменять нельзя, поэтому исполнение следующего примера приводит к ошибке:

```
>>> s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> s[1] = "K"
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  TypeError: object doesn't support item assignment
>>> # ошибка: объекту не поддерживает присваивание своим элементам
```

Но вы можете добиться желаемого результата, создав новую строку:

```
>>> s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> s = s[0] + "K" + s[2:]
>>> print s
AKCDEFGHIJKLMNOPQRSTUVWXYZ
```

А в следующем примере происходит подсчет количества вхождений одной строки в другую:

```
>>> import string
>>> text = "Мы, значит, поехали, значит, на поезде, значит..."
>>> print "Повторов слова 'значит':", string.count(text, "значит")
Повторов слова 'значит': 3
```

Модуль `string`, из которого мы взяли функцию `count`, имеет много других функций для работы со строками *(см. разд. 9.3.1)*.

В следующем примере мы находим позиции, в которых одна строка встречается в другой:

```
>>> import string
>>> s = "Все тот же пример."
>>> string.find(s, "пример")
11
>>> string.find(s, "строка")
-1
```

В первом случае мы нашли, что строка `"пример"` встречается, начиная с одиннадцатой позиции. Во втором случае строка не найдена вообще: воз-

вращается -1. Чтобы найти последующие вхождения одной строки в другую, можно задать третий аргумент функции find(), который указывает, с какого места производить поиск:

```
>>> import string
>>> s = "Все тот же пример."
>>> string.find(s, "т")
4
>>> string.find(s, "т", 5)
6
```

Упражнение

Напишите функцию для вырезания из строки первого слова (считать, что между словами стоят пробелы).

В строке можно производить замены одной подстроки на другую из некоторой строки:

Листинг 2.6. Замена

```
>>> import string
>>> text = "Пример пременения замен, простой для воспреятия."
>>> print string.replace(text, "пре", "при")

Пример применения замен, простой для восприятия.
```

Заметьте, что При и при являются разными строками и, естественно, что замена не затрагивает При.

По мере надобности, мы изучим и другие возможности по обработке строк, но и сейчас мы знаем о строках достаточно, чтобы решать конкретные задачи.

2.5. Условия

Никакая более или менее сложная программа не может обойтись без принятия решений. До сих пор наши программы исполнялись от начала и до конца последовательно, оператор за оператором. С помощью оператора if (если) можно заставить выполнение программы пойти по одному из альтернативных путей. Выбор пути зависит от выполнения заданного в операторе условия. Схематично if можно изобразить так:

```
if условие:
  первый фрагмент
else:
  второй фрагмент
```

Если `условие` истинно, будет выполнен первый фрагмент, иначе — второй (слово *else* по-англ. иначе). Часть `else` оператора `if` может и отсутствовать. Тогда в случае, если условие ложно, не будет выполняться ничего.

Рассмотрим простой пример:

```
a = input()                        # ввод a
if a >= 0:                         # если a >=0, то:
  print "a -- неотрицательное"     #   печатать "a -- неотрицательное"
else:                              # иначе:
  print "a -- отрицательное"       #   печатать "a -- отрицательное"
print "Вот так"                    # печатать "Вот так"
```

В этом примере программа скажет `"a -- неотрицательное"` если выполнено условие, стоящее в операторе `if`. Иначе программа напишет: `"a -- отрицательное"`. Таким образом, ход выполнения программы пойдет только по одному руслу, в зависимости от значения `a`. Затем оба эти русла сольются в одно, и программа напечатает `"Вот так"`.

В этой книге оператор `if` назван условным, но иногда его называют оператором ветвления, чтобы подчеркнуть его роль в программе.

Приведенная выше программа кажется любителям Pascal и C несколько незаконченной. Не совсем понятно, как Python определяет, где заканчиваются части оператора `if` — ведь нет никаких `begin-end` или даже скобок! Оказывается, все дело в отступах. Отступы — единственный способ указать границы блока в Python. Таким образом, красиво расположенный программный код в Python — не роскошь, а требование интерпретатора.

В программе может быть много условных операторов, они могут быть вложенными один в другой. Важно только, чтобы операторы, записанные внутри оператора `if`, имели больший отступ от левого края, чем сам оператор `if`. Так в языке Python выделяются группы операторов.

Примечание

В других языках программирования для выделения операторов в группу используются скобки или слова, а отступы необязательны, но помогают лучше видеть структуру программы.

В следующем примере мы напишем функцию, которая просклоняет слово "рубль" в зависимости от заданной суммы. Проанализировав задачу, мы видим, что в первой двадцатке бывают три ситуации: 0 и 5—20 рублей, 1 рубль, 2—4 рубля. Дальше до 100 все определяет последняя цифра: *0, *5—9 рублей,

*1 рубль, *2—4 рубля. Для больших чисел видно, что на склонение влияют только две последние цифры числа, можно составлять алгоритм:

```
функция Склонение_Рубля(x):
если предпоследняя цифра равна 1:
  возвратить "рублей"
иначе:
  если последняя цифра равна 1:
    возвратить "рубль"
  иначе:
    если последняя цифра 2, 3 или 4:
      возвратить "рубля"
    иначе:
      возвратить "рублей"
```

Последнюю цифру в десятичной записи числа можно получить, взяв остаток от деления числа на 10 (в Python получение остатка от деления делается с помощью операции `%`). Предпоследнюю цифру можно получить, если сначала нацело поделить число на 10 (при этом предпоследняя цифра станет последней) и поступить как с последней цифрой. Сравнения чисел в Python можно записывать как в математике, цепочкой, а проверка равенства чисел записывается двойным знаком равно "==". Теперь у нас есть все, чтобы написать программу:

Листинг 2.7. Склонение слова "рубль"

```
def rouble(x):
  if (x/10) % 10 == 1:       # предпоследняя цифра равна 1?
    return "рублей"
  else:
    if x % 10 == 1:          # последняя цифра равна 1?
      return "рубль"
    else:
      if 2 <= x % 10 <= 4:   # последняя цифра от 2 до 4?
        return "рубля"
      else:
        return "рублей"

# проверяем:
print 10, rouble(10), 21, rouble(21), 104, rouble(104)
print 14, rouble(14), 30, rouble(30), 1138, rouble(1138)
```

Здесь мы видим, что очень часто после `else` идет `if`. В языке Python для этого случая есть `elif`. Следующий вариант программы использует оператор `elif`:

Листинг 2.8. Склонение слова "рубль" (дубль 2)

```
def rouble(x):
  if (x/10) % 10 == 1:
    return "рублей"
  elif x % 10 == 1:
    return "рубль"
  elif 2 <= x % 10 <= 4:
    return "рубля"
  else:
    return "рублей"
print 10, rouble(10), 21, rouble(21), 104, rouble(104)
print 14, rouble(14), 30, rouble(30), 1138, rouble(1138)
```

Программу стало удобнее читать, не правда ли? А результат работы тот же. Такой оператор с частями `elif` уже не просто условный оператор, а оператор выбора. В операторе выбора условия проверяются по порядку, пока не найдется выполняющееся. И только одна часть этого оператора исполняется (либо, если `else` нет вообще, не исполняется никакая).

> ***Упражнение***
>
> В качестве упражнения читателю предлагается написать программу для склонения слова "копейка".

2.6. Немного о логике

Современный электронный компьютер построен и работает на строгой логической основе. Учеными разработано много различных логик, но нас будет интересовать булева логика (по имени английского математика XIX века Джорджа Буля). Эта логика работает над двумя значениями, которые можно назвать "истина" и "ложь" и кратко обозначить как `1` и `0`. (Язык Python понимает под "ложью" не только `0`, но и, например, строку нулевой длины.) В логических формулах наиболее часто используются операции отрицания (логическое НЕ), *дизъюнкции* (логическое ИЛИ) и *конъюнкции* (логическое И).

Отрицание (в языке Python оно обозначается как `not`) работает очень просто. Значение "истина" становится "ложью", а "ложь" — "истиной".

```
not 0 -> 1
not 1 -> 0
```

Дизъюнкция (обозначается `or`) принимает два аргумента, и ее результат "ложь" только если оба аргумента "ложь". Можно заметить, что логическое ИЛИ отличается от бытового понимания и скорее походит на "канцелярский" союз "и/или":

```
0 or 0 -> 0
0 or 1 -> 1
1 or 0 -> 1
1 or 1 -> 1
```

Конъюнкция (`and`) двух аргументов — "истина" только в том случае, если оба аргумента "истины":

```
0 and 0 -> 0
0 and 1 -> 0
1 and 0 -> 0
1 and 1 -> 1
```

Замечание

Булева логика не может ответить на вопрос об истинности того или иного утверждения. Она может лишь на основе правильных входных данных и правильно составленной формулы дать правильный ответ.

Вернемся к примеру из предыдущего раздела. Теперь мы видим, что слово `"рублей"` возвращается в двух случаях. А нельзя ли подвести эти случаи под одно условие? Можно. Но придется вспомнить условие для последней части. Вот оно:

```
x%10 == 0 или 5 <= x%10 <= 9
```

Учитывая еще и условие для предпоследней цифры, получаем:

Листинг 2.9. Склонение слова "рубляь" (дубль 3)

```
def rouble(x):
  if (x/10)%10 == 1 or x%10 == 0 or 5 <= x%10 <= 9:
    return "рублей"
  elif x % 10 == 1:
    return "рубль"
  else:
    return "рубля"

print 10, rouble(10), 21, rouble(21), 104, rouble(104)
print 14, rouble(14), 30, rouble(30), 1138, rouble(1138)
```

Ответ прежний:

```
10 рублей 21 рубль 104 рубля
14 рублей 30 рублей 1138 рублей
```

В математической логике определяются и правила, позволяющие переходить от одной формулы к другой. В табл. 2.1 указаны лишь некоторые из них.

Таблица 2.1. *Некоторые правила работы с логическими формулами*

Исходная формула	Логически тождественная ей формула
`not not A`	`A`
`A and (not A)`	`0`
`A or (not A)`	`1`
`not (A and B)`	`(not A) or (not B)`
`not (A or B)`	`(not A) and (not B)`
`A and (B or C)`	`(A and B) or (A and C)`
`A or (B and C)`	`(A or B) and (A or C)`

На практике применение формул выглядит так. Вместо не совсем удобной формулы `not (A < 5 or A > 10)` мы можем записать: `A >= 5 and A <= 10`, что с помощью Python можно записать и еще более красиво: `5 <= A <= 10`.

Упражнение

Попробуйте с помощью изученных правил избавиться от оператора `not` в выражении `not((x/10)%10 == 1 or x%10 == 0 or 5 <= x%10 <= 9)`.

2.7. Циклы

Основное преимущество компьютеров перед человеком состоит в том, что они могут без устали секунда за секундой, год за годом выполнять одни и те же действия. В этом разделе мы рассмотрим *цикл* — конструкцию, которая выполняет один и тот же участок программы (называемый *телом цикла*) нужное число раз. Каждое такое повторение часто называют *итерацией*.

В Python имеются два вида операторов цикла: *цикл while (пока)* и *цикл for (для)*.

Первый цикл производит некоторые действия, пока выполняется заданное условие. В Python этот вид цикла записывается следующим образом:

```
while условие:
  тело цикла
```

В данном случае, пока `условие` истинно, будет выполняться снова и снова тело цикла. Правильно составлять условия мы уже научились в предыдущем разделе.

В листинге 2.10 приведена программа игры "Угадай число". Компьютер загадывает случайное число от 1 до 100, а человек пытается его отгадать, вводя свои предположения.

Листинг 2.10. Файл guess_number.py. Игра "Угадай число"

```
from whrandom import randint
max_number = 100   # по какое число загадывать
my_number = randint(1, max_number)  # загадываем число
print "Число от 1 до ", max_number, "загадано."
guess = None
tries = 0
while guess != my_number:  # пока число не угадано:
  tries = tries + 1          # увеличиваем счетчик попыток на 1
  print "Попытка", tries
  guess = int(raw_input("Введите ваш вариант: "))
  if my_number < guess:
    print "Меньше!"
  elif guess < my_number:
    print "Больше!"
print "Число угадано за", tries, "попыток (попытки)."
```

Типичный диалог с программой выглядит так:

```
Число от 1 до 100 загадано.
Попытка 1
Введите ваш вариант: 50
Больше!
Попытка 2
Введите ваш вариант: 75
Меньше!
Попытка 3
Введите ваш вариант: 60
Меньше!
Попытка 4
Введите ваш вариант: 54
Число угадано за 4 попыток (попытки).
```

В этом примере цикл `while` работает, пока пользователь вводит неверные догадки, т. е. загаданное число `guess` не равно введенному пользователем `my_number`. Строки, которые повторяются в цикле, отступают на два пробела по сравнению с другими. Последний оператор программы — оператор `print` — не имеет отступа, и значит, находится вне цикла, сразу за ним.

В этой же программе нам встретились новые функции. Функция `randint()` при каждом своем вычислении дает так называемое *псевдослучайное число* из заданного интервала. `raw_input()` вводит строку, а `int()` получает из строки целое число. Конечно, вместо применения двух этих функций можно было бы применить `input()`, но в этом случае пользователь смог бы сыграть нечестно, просто введя на запрос компьютера `guess`.

Упражнение

Попробуйте усложнить программу, введя фиксированное число попыток для угадывания числа.

Другой вид цикла — цикл `for` — можно схематично записать так:

```
for параметр in последовательность:
  тело цикла
```

Можно увидеть, что в цикле `for` применяется специальная переменная, которую обычно называют параметром цикла. В языке Python параметр цикла `for` пробегает все значения из заданной последовательности. *(О последовательностях мы поговорим в разд. 2.9.)*

Пусть нам необходимо посчитать площади кругов от 1 до 10. В этом случае мы можем применить написанную ранее функцию `Square()` внутри цикла:

```
import math
def Square(radius):
  return math.pi * radius**2


for r in xrange(1, 11):
  print "Площадь круга радиуса", r, "равняется", Square(r)
```

Результат выполнения:

```
Площадь круга радиуса 1 равняется 3.14159265359
Площадь круга радиуса 2 равняется 12.5663706144
Площадь круга радиуса 3 равняется 28.2743338823
Площадь круга радиуса 4 равняется 50.2654824574
Площадь круга радиуса 5 равняется 78.5398163397
Площадь круга радиуса 6 равняется 113.097335529
Площадь круга радиуса 7 равняется 153.938040026
```

```
Площадь круга радиуса 8 равняется 201.06192983
Площадь круга радиуса 9 равняется 254.469004941
Площадь круга радиуса 10 равняется 314.159265359
```

В качестве параметра цикла в указанном примере выступает переменная `r`, а значения, которые он пробегает, заданы с помощью функции `xrange()`. Здесь сто́ит заметить, что в этой функции в качестве аргумента задано число 11, тогда как цикл выполняется лишь до 10 включительно. Просто в Python так принято во многих случаях: интервал считается от чего-то (включительно) до чего-то (исключительно). В наших примерах `xrange` порождает последовательности чисел специально для цикла `for`.

Следующий пример иллюстрирует возможность организации вложенных циклов:

```
from whrandom import randint
for i in xrange(1, 8):
  for j in xrange(i, 8):
    print randint(0, 1),
  print
```

Результат выполнения:

```
0 1 1 1 1 1 0
1 0 1 0 0 1
0 1 1 0 1
0 1 0 0
1 1 0
1 0
0
```

Заметьте, что внутренний цикл при каждом значении `i` выполняется `8-i` раз. Таким образом, когда `i` равно 1, цикл `for j` выполнится семь раз, выведя семь случайных чисел (0 или 1). При последнем значении параметра `i`, когда тот равен 7, внутренний цикл выполнится лишь один раз.

Вложенность операторов выделена отступами. Оператор `print` без параметров относится к внешнему циклу и просто выводит признак конца строки. Другой оператор `print` принадлежит внутреннему циклу. Попутно заметим, что если после последнего параметра в операторе `print` стоит запятая, то перевода строки не происходит. Благодаря этому числа выводятся через пробелы.

Упражнение

Напишите программу для вывода таблицы умножения.

Если ваш Python установлен полностью, вы можете изучать циклы с помощью... черепашки как в языке Logo. В следующем примере мы нарисуем несколько квадратов.

Листинг 2.11. Управление черепашкой

```
import turtle     # импортируем черепаху
turtle.reset()    # готовим черепаху к выполнению команд
for i in xrange(6):   # будем рисовать шесть квадратов
  turtle.down()          # черепаха, опусти перо
  for j in xrange(4):   # рисуем четыре стороны квадрата
    turtle.forward(10)     # черепаха, вперед 10
    turtle.left(90)        # черепаха, поворот на 90 градусов
  turtle.up()             # черепаха, подними перо
  turtle.forward(20)      # переходим к следующему окну
turtle.left(180)     # поворот на 180 градусов
turtle.forward(140)  # уводим черепаху
turtle._root.mainloop()  # команда, удерживающая черепашье окно открытым
```

Результат можно увидеть на рис. 2.2.

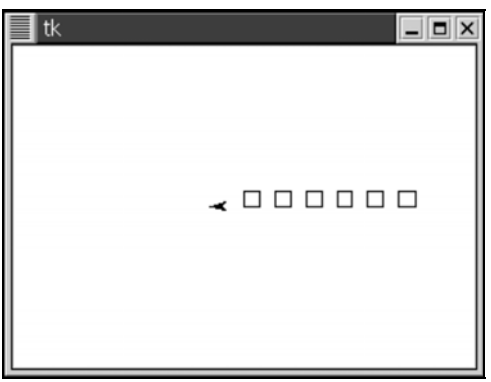


Рис. 2.2. Шесть квадратов

Упражнение

Напишите программу для рисования квадратов в три ряда.

2.8. Пишем комментарии

Мы уже много раз видели решеточки "#" в текстах программ. Они говорят интерпретатору Python, что до конца строки идет комментарий, и его нужно

игнорировать. Комментарии пишутся для людей. Иногда приходится возвращаться к давно написанной программе, и подчас очень сложно разобраться, что же мы имели в виду. Никогда не экономьте на комментариях! В реальных программах комментарии должны объяснять не язык программирования (предполагается, что читающий программу знает язык), а саму программу. Пример:

Листинг 2.12. Хорошо прокомментированная программа

```
print "Введите радиусы нескольких кругов через запятую:",
krugi = input()
for r in krugi:       # для всех радиусов
  if 1 <= r <= 10:    # если радиус подходящий
    S = 3.14159 * r**2   # вычисляем площадь круга и выводим:
    print "Площадь круга радиуса", r, "равняется", S
  else:               # иначе пропускаем вычисление
    print "Пропускаем круг радиуса", r
```

Надеемся, что в этом примере все понятно благодаря комментариям.

2.9. Учимся использовать списки

Язык Python имеет хорошую поддержку для последовательностей. Один из видов последовательностей мы уже знаем — это последовательность символов, т. е. строка. Помимо строки, в Python есть *список* (изменяемая последовательность элементов) и *кортеж* (неизменяемая последовательность элементов).

Список — это просто последовательность элементов, которую можно изменять. Списки встречаются в нашей жизни очень часто. Это — расписание, план, перечень предметов, список сотрудников, меню, содержание книги. Даже алгоритм (программу) можно назвать списком действий. Списки бывают простые, состоящие из однотипных элементов, и сложные, содержащие в качестве своих элементов другие списки (как, например, пункты сложного плана). Отличие кортежей от списков состоит в том, что их состав нельзя изменять, а записываются они в круглых (а не квадратных) скобках. Синтаксис задания списков и кортежей следующий:

```
кортеж: (элемент1, элемент2, ..., элементN,)
список: [элемент1, элемент2, ..., элементN,]
```

Познакомиться со списками и кортежами лучше с помощью примеров. В следующем примере показаны списки и кортежи различной длины:

Листинг 2.13. Примеры записи списков и кортежей

```
empty_list = []
empty_tuple = ()
one_el_list = ["a"]
one_el_tuple = ("a",)
some_list = [1, "b", 12.45]
some_tuple = ("Red", "Green", "Blue")
```

Как и другие последовательности, списки можно соединять друг с другом:

```
>>> a = [1, 2]
>>> b = [7, 8, 9]
>>> print a+b
[1, 2, 7, 8, 9]
```

Списки (и кортежи) могут быть вложенными друг в друга:

```
>>> nested_tuple = (1, (2, 3))
>>> nested_list = [1, [2, [3, 4], 5], 6]
>>> mixed_list = [1, (2, 4), "four", 5.0, ["a", "b"]]
>>> print len(mixed_list)
5
```

Здесь `nested_tuple` — кортеж из двух элементов. Первый элемент `1`, а второй — другой кортеж `(2, 3)`. Аналогично `nested_list` является списком из трех элементов, второй из которых является тоже списком. Наконец, `mixed_list` является списком различных объектов, показывая, что в списке может храниться все, что угодно. Длина списка `mixed_list` равна пяти, что мы легко узнаем с помощью уже знакомой нам функции `len`.

Упражнение

Как на языке Python может выглядеть список координат трех точек на плоскости?

По аналогии со строками, из последовательности можно вырезать элементы по индексам:

```
>>> spisok = [1, 2, 3, 4, 5]
>>> print spisok[3]
4
>>> print spisok[-1]
5
>>> print spisok[:2]
[1, 2]
```

Напоминаем, что первый элемент — нулевой в Python, т. к. индексируются не сами элементы, а промежутки между ними. Запись `[n:n]` означает `n`-й промежуток, т. е. промежуток между `n-1`-ым и `n`-ым элементами. Запись `[1:3]` в последующем примере выделяет первый и второй элемент по нумерации Python (то есть второй и третий по счету). Третий элемент в операциях не участвует. Пример в листинге 2.14 показывает, как различные операции влияют на содержимое списка. Обратите внимание, что аргументом `remove()()` является не индекс, а значение элемента.

Листинг 2.14. Работа со списком

```
>>> spisok = [1, 2, 3, 4, 5, 1]
>>> print "Количество единиц:", spisok.count(1)
Количество единиц: 2
>>> spisok.append(4)    # добавляем элемент к списку
>>> print "После добавления:", spisok
После добавления: [1, 2, 3, 4, 5, 1, 4]
>>> spisok.remove(4)    # убираем первый элемент с заданным значением
>>> print "После удаления:", spisok
После удаления: [1, 2, 3, 5, 1, 4]
>>> spisok[1:1] = [8, 9]  # вклиниваем два элемента
>>> print "После вклинивания:", spisok
После вклинивания: [1, 8, 9, 2, 3, 5, 1, 4]
>>> spisok[1:3] = [7]
>>> print "После замены:", spisok
После замены: [1, 7, 2, 3, 5, 1, 4]
>>> spisok[1] = 0
>>> print "После замены элемента:", spisok
После замены элемента: [1, 0, 2, 3, 5, 1, 4]
```

Если вы проделали все действия вслед за интерпретатором, то вам должно быть ясно, почему в последовательностях индексируются не сами элементы, а промежутки между ними. Это сделано для того, чтобы в операциях вроде `spisok[1:1] = [8, 9]` было удобно задавать место, куда добавляются новые элементы списка.

Упражнение

Как можно осуществить `spisok.append(x)` без применения метода `append`?

С помощью функций из модуля `string` строки можно разбивать (split) на список строк, а список строк можно объединять (join) опять в одну строку:

```
>>> import string  # загружаем модуль для работы со строками
>>> s = string.split("1,2,3,4,5", ",")
>>> print s
['1', '2', '3', '4', '5']
>>> ss = string.join(s, "; ")
>>> print ss
1; 2; 3; 4; 5
```

Упражнение

Напишите программу, которая принимает текст в одной строке и печатает его по одному предложению в строке.

Следующий веселый пример показывает, как списки используются в качестве последовательностей в цикле `for`:

Листинг 2.15. Составляем слова

```
ps = ["при", "за", "от"]
ks = ["корм", "лож", "точ"]
ss = ["", "енный", "ка"]
for p in ps:
  for k in ks:
    for s in ss:
      print p+k+s,
    print
```

Результат выполнения:

```
прикорм прикорменный прикормка
прилож приложенный приложка
приточ приточенный приточка
закорм закорменный закормка
залож заложенный заложка
заточ заточенный заточка
откорм откорменный откормка
отлож отложенный отложка
отточ отточенный отточка
```

2.10. Работаем с текстом

Даже если судить по уже изученным возможностям, Python прекрасно подходит для обработки текстовой информации. В первом примере мы увидим, как можно над текстом произвести простейшую операцию замены одной подстроки на другую. Этот пример с подготовкой текста телеграммы — скорее, повторение того, что мы уже знаем, но он призван показать, что текст в Python можно представить в виде одной (хоть и длинной) строки:

```
import string
text = """Python считает словами все, что разделено пробельными
символами. К пробельным относятся не только пробелы,но и знаки
табуляции, а также возврат каретки."""
text1 = string.replace(text, ",", " зпт ")
text2 = string.replace(text1, ".", " тчк ")
print text2
```

Результат выполнения:

```
Python считает словами все зпт  что разделено пробельными
символами тчк  К пробельным относятся не только пробелы зпт но и знаки
табуляции зпт а также возврат каретки тчк
```

> ***Упражнение***
>
> Напишите программу для замены предлогов "в", "на", "к" на пробелы и дополните приведенный пример.

Предположим, что нам нужно посчитать все слова в тексте и вывести их. Текст для удобства изложения приведен в самой программе:

Листинг 2.16. Считаем слова в тексте

```
from string import split  # из модуля работы со строками нужна
                          # только функция split
text = """Python считает словами все, что разделено пробельными
символами. К пробельным относятся не только пробелы,но и знаки
табуляции, а также возврат каретки."""
words = split(text)       # разбить текст на слова
print "Слов в тексте:", len(words)
for word in words:
  print word
```

Результат выполнения:

```
Слов в тексте: 21
Python
считает
словами
все,
что
разделено
пробельными
символами.
К
пробельным
относятся
не
только
пробелы,но
и
знаки
табуляции,
а
также
возврат
каретки.
```

Как видно из приведенного примера, слова подсчитаны неправильно. Виной тому — неаккуратный ввод текста: "`пробелы,но`". Конечно, мы можем для частного случая (подсчет слов) заменить все небуквенные символы, скажем, на пробелы и повторить подсчет. Но гораздо полезнее познакомиться с основами работы с так называемыми регулярными выражениями и применять их для решения задач обработки текста. Но об этом в следующем разделе.

2.11. Знакомимся с регулярными выражениями

При работе с текстами часто встречаются ситуации, когда искомые данные имеют некоторый вполне определенный формат. Например:

- пункты плана начинаются с начала строки, и после числа, обозначающего номер пункта, обычно идет точка, скобка или их сочетание;

- электронное письмо состоит из двух частей — шапки и собственно самого текстового сообщения. В шапке в виде полей с различными именами среди прочего даны тема письма, получатель, отправитель, срочность доставки, дата и т. п.;
- экспортированные из электронной таблицы данные могут иметь форму текста, где в каждой строчке записываются значения клеток таблицы через некоторый разделитель, например, запятую;
- в программе на языке программирования функции записываются согласно определенному в этом языке синтаксису. Например, в Python имя определяемой функции следует после ключевого слова `def`, а затем в скобках записываются аргументы.

Во всех этих примерах данные, находящиеся в тексте, можно найти по характерным признакам. Для фиксации этих признаков и служат *регулярные выражения*.

Регулярное выражение — формула, которую можно применять для поиска подстрок определенного вида в данной строке. Регулярное выражение можно считать шаблоном, описывающим сразу целый набор различных строк. В предыдущем разделе мы производили поиск точно известной подстроки и ее замену на другую подстроку (мы знали, что нужно заменить `","` на `"зпт"`). Благодаря большóй описательной мощности, с помощью регулярных выражений можно решать гораздо более сложные задачи.

Функции для работы со строками с помощью регулярных выражений находятся в модуле по имени `re` (regular expressions).

В регулярных выражениях используются специальные обозначения, полный список и детальное описание которых желающие могут посмотреть в *разд. 9.3.2*. Для знакомства нам хватит набора обозначений, приведенных в табл. 2.2.

Таблица 2.2. *Некоторые символы, имеющие особое значение в регулярных выражениях*

Символ	Что обозначает
`"."`	Любой символ
`"^"`	Начало строки
`"$"`	Конец строки
`"*"`	Повторение фрагмента ноль или более раз
`"+"`	То же, но хотя бы один раз
`"?"`	Предыдущий фрагмент либо присутствует, либо отсутствует
`"{m,n}"`	Повторение предыдущего фрагмента от `m` до `n` раз включительно

Таблица 2.2 *(окончание)*

Символ	Что обозначает
`"[...]"`	Любой символ из набора в скобках
`"[^...]"`	Любой символ не из набора в скобках
`"\"`	С помощью символа обратной косой черты можно отменять специальное значение следующего за ней символа, например `"\("` — соответствует `"("`
`"|"`	Соответствует символам справа или слева
`"(...)"`	Скобки, внутри которых находится регулярное выражение

Помимо специальных обозначений, в регулярном выражении только буквы и цифры могут входить в качестве самих себя. Для внесения других символов необходимо использовать обратную косую черту. Например, скобку можно задать как `"\("`.

С регулярным выражением можно не только сопоставлять строки (подходит — не подходит), но и искать строки, удовлетворяющие регулярному выражению, в тексте. В табл. 2.3 приведены регулярные выражения и примеры сопоставимых с ними и не сопоставимых с ними строк.

Таблица 2.3. *Примеры регулярных выражений*

Запись шаблона	Примеры строк, удовлетворяющих шаблону	Примеры строк, не удовлетворяющих шаблону	Примечание
`".*"`	`"любая строка"`		Произвольное число раз повторяется любой символ
`"[abc]+"`	`"a"`, `"baba"`, `"abba"`	`""`, `"adidas"`	Непустая строка из символов `a`, `b`, `c`
`"[a-z]+"`	`"aakkaa"`, `"zyx"`, `"code"`	`""`, `"Abba"`, `"ADIDAS"`	Непустая строка из строчных латинских букв
`"[a-h][1-8]"`	`"e2"`, `"e4"`, `"h1"`	`"k1"`, `"h0"`, `"a"`	Координаты клетки шахматной доски
`"^From: .*"`	`"From: abc@mail.ru"`	`" From: nospam"`	Поле электронного письма
`"^[A-Z][a-z]*$"`	`"One"`, `"Name"`	`"Two names"`, `"other"`	Строка с записью одного слова латинскими буквами, с большой буквы
`"A?B?C?"`	`""`, `"A"`, `"B"`, `"C"`, `"AB"`, `"AC"`, `"BC"`, `"ABC"`	`"BA"`, `"AA"`, `"A?B?C?"`	В столбце примера приведены все строки, которые удовлетворяют шаблону

Таблица 2.3 *(окончание)*

Запись шаблона	Примеры строк, удовлетворяющих шаблону	Примеры строк, не удовлетворяющих шаблону	Примечание
`"\([0-9]{2,4}\)"`	`"(11)"`, `"(9999)"`, `"(345)"`	`"(1)"`, `"()"`, `"(12345)"`, `"\(111\)"`	Двух-четырех значное число в скобках
`"(яблоко|апельсин)"`	`"яблоко"`, `"апельсин"`	`"апелько"`, `"яблокоапельсин"`	Либо `"яблоко"`, либо `"апельсин"`
`"(яблоко | апельсин )+"`	`"яблоко "`, `"апельсин яблоко апельсин "`, `"апельсин "`	`""`, `"апельсин"`, `"яблокоапельсин"`	Положительное число повторений регулярного выражения в скобках
`"(кис+)+"`	`"кис"`, `"кискискис"`, `"киссс-скисскисссссс"`	`"киисс"`	`"кис"` несколько раз с любым числом букв `"с"` на концах
`"[^():,.-]+"`	`"слова"`, `"слова и пробелы"`	`"-"`, `"(скобки)"`, `"запятые,"`	Все, кроме указанных после `"^"` знаков. Заметьте, что `"-"` может стоять только в конце `"[...]"`, чтобы обозначать саму себя

В следующей программе все числа в строке заменяются звездочками:

```
import re
text = """10, 12; 123 (127)/2"""
text1 = re.sub("[0-9]+", "*", text)
print text1
```

Результат выполнения:

```
*, *; * (*)/*
```

Как легко увидеть из примера, функция `re.sub()` играет ту же роль, что и `string.replace()`, но вместо одной-единственной подстроки позволяет заменять на `"*"` любые подстроки, которые соответствуют заданному регулярному выражению `"[0-9]+"`.

Упражнение

Напишите регулярные выражения для поиска в тексте адресов электронной почты и замены их на `"@@@"`.

А в следующем примере с помощью функции `search()` мы найдем, кому адресовано электронное письмо:

```
import re
letter = """To: me@mail.ru
From: you@somewhere.net
Subject: просто письмо

Привет! Это я.

You."""
m = re.search("^To: (.+)$", letter, re.MULTILINE)
print m.group(1)
```

Результат выполнения:

```
me@mail.ru
```

В регулярном выражении `"^To: (.+)$"` искомое выделено с помощью скобок. Остальное является контекстом, в котором должно находиться то, что мы ищем. Здесь нам пригодились `"^"` и `"$"`, отмечающие начало и конец строки. Флаг `re.MULTILINE` добавлен для указания того, что `"^"` и `"$"` отмечают начало и конец любой строки, а не только всего текста. Результат поиска в виде специального объекта записывается в `m`, после чего мы можем получить найденный в первой группе скобок результат, написав `m.group(1)`. Конечно, в данном примере предполагается, что нужный нам фрагмент встречается ровно один раз (в письме не может быть двух полей `To`).

Следующий пример находит все поля электронного письма и их значения с помощью функции `re.findall()`:

```
import re
letter = """From: you@somewhere.net
To: me@mail.ru
Subject: просто письмо

Привет! Это я.

You."""
pattern = re.compile("^([a-zA-Z_-]+): (.+)", re.M)
for (field, value) in pattern.findall(letter):
  print field, "=", value
```

Результат выполнения:

```
From = you@somewhere.net
To = me@mail.ru
Subject = просто письмо
```

В этой программе мы сначала подготовили ("скомпилировали") регулярное выражение с помощью функции `re.compile()` в специальный объект (в нашей программе он назван `pattern`). Функции `re.compile()` передан и второй аргумент — флаг `re.M`, синоним `re.MULTILINE`. Затем использован его метод `findall()` для нахождения всех полей и их значений. *(Об объектах и методах мы узнаем несколько позже, в гл. 5.)* Метод `findall()` дает список списков найденных по шаблону групп символов. В нашем регулярном выражении всего две группы символов (в регулярном выражении они выделены скобками), поэтому элементы кортежа `(field, value)` последовательно получают имена полей и их значения, которые в целях иллюстрации просто выводятся с помощью `print`.

Упражнение

Какая подстрока будет первой найдена в строке "`a = (b + c) * 10`" при использовании регулярного выражения "`[a-z] [=+*] [a-z]`"?

Теперь можем вспомнить задачу из предыдущего раздела, где мы пытались выделить из строки список слов. Задачу разбиения текста на слова можно решить, задав разделитель с помощью регулярного выражения "`[ .,;()?!\n-]+`", т. е. повторяющиеся один или несколько раз пробелы, признак конца строки ("`\n`") и знаки пунктуации. Правда, при этом возникает проблема со строками, начинающимися или оканчивающимися разделителем, т. к. в получаемом списке слов оказываются лишние "слова" — пустые строки. Чтобы этого избежать, можно проверить первый и последний элементы списка `words`, удалив их, если они пустые.

Листинг 2.17. Снова считаем слова в тексте

```
from re import split  # берем split из модуля re
                      # для работы с регулярными выражениями
text = """Python считает словами все, что разделено пробельными
символами. К пробельным относятся не только пробелы,но и знаки
табуляции, а также возврат каретки."""
words = split("[ .,;()?!\n-]+", text) # разбить текст на слова
if words[0] == "":       # если в начале списка пустое слово
  del words[0]           # удаляем его
if words[-1] == "":      # то же для последнего элемента списка
  del words[-1]
print "Слов в тексте:", len(words)
for word in words:
  print word
```

Результат не обманул наших ожиданий:

```
Слов в тексте: 22
Python
считает
словами
все
что
разделено
пробельными
символами
К
пробельным
относятся
не
только
пробелы
но
и
знаки
табуляции
а
также
возврат
каретки
```

Помимо функции `split()`, в модуле `re` есть такие функции, как `search()` (искать), `sub()` (substitute, подставить) и некоторые другие, о которых вы можете узнать в *разд. 9.3.2*.

> ***Упражнение***
>
> Напишите программу для выделения из текста всех предложений и подсчета их длин.

2.12. Словари

В жизни мы часто пользуемся *словарями* — это и телефонный справочник, в котором по фамилии, инициалам и адресу мы находим телефон, и меню программы, где нужное действие выбирается указанием соответствующей фразы, и, конечно, собственно словарь — книга, в которой зная слово (ключ),

мы находим (в зависимости от назначения словаря) варианты перевода слова на другой язык, толкование слова и т. п.

В языке Python словарь (dictionary) — отображение между ключами (keys) и значениями (values), при котором ключу однозначно соответствует значение. В других языках программирования словари могут называться другими именами: хэши, ассоциативные массивы. В словаре не может быть двух одинаковых ключей, тогда как на значения таких ограничений не накладывается. Синтаксис для задания словаря следующий:

```
словарь: {ключ1: значение1, ключ2: значение2, ..., ключN: значениеM,}
```

В приведенном ниже примере вы увидите, как задать словарь и как работать с его содержимым.

```
>>> dict = {"one": "раз", "two": "два", "three": "три"}
>>> print 'Берем значение по ключу "one":', dict["one"]
Берем значение по ключу "one": раз
>>> dict["zero"] = "ноль"  # записываем значение для ключа "zero"
>>> dict["zero"] = "нуль"  # то же (старое значение переписывается)
>>> print dict["zero"]
нуль
>>> del dict["zero"]  # удаляем значение для ключа "zero" из словаря
```

Итак, словарь задается внутри скобок `{}`. Пары ключ-значение разделяются запятыми. Между ключом и значением ставится `":"`. Ключу можно присвоить новое значение, при этом старое значение теряется (если оно было). Доступ к значению по ключу записывается так же, как доступ к элементу списка по его номеру.

Узнать, имеется ли некоторый ключ в словаре, можно с помощью метода `has_key`:

```
if dict.has_key("zero"):  # проверяем наличие ключа
  print 'dict имеет ключ "zero"'
else:
  print 'dict не имеет ключа "zero"'
```

В следующем примере мы увидим, как получить список ключей, значений и список кортежей с ключами и значениями (`keys()`, `values()` и `items()`):

Листинг 2.18. Циклы по ключам и значениям словаря

```
dict = {"one": "раз", "two": "два", "three": "три"}
print "   Выводим все ключи:"
for k in dict.keys():
  print k,
print "\n   Выводим все ключи и значения:"
```

```
# Выводим все значения:
for v in dict.values():
  print v,
print "\n   Выводим все ключи и значения:"
for (k, v) in dict.items():
  print k, ":", v, ",",
```

Результат выполнения:

```
   Выводим все ключи:
one two three five four
   Выводим все ключи и значения:
один два три пять четыре
   Выводим все ключи и значения:
one : один , two : два , three : три , five : пять , four : четыре ,
```

В данном примере мы сразу используем списки в цикле `for`. Заметьте, что метод `items()` возвращает список двухэлементных кортежей, поэтому мы можем в качестве параметра цикла `for` указать сразу две переменные, `k` и `v`, которые одновременно получают значения при каждом повторении цикла.

Примечание

Порядок следования элементов в словаре не имеет значения, поэтому при выводе `"five"` появилось ранее, чем `"four"`.

```
>>> dict = {"one": "раз", "two": "два"}
>>> dict2 = {"one": "один", "four": "четыре", "five": "пять"}
>>> dict.update(dict2)  # обновить словарь dict из словаря dict2
>>> print 'dict["one"], dict["five"]:', dict["one"], dict["five"]
dict["one"], dict["five"]: один пять
```

В этом примере значения из словаря `dict2` переписываются в словарь `dict`, затирая старые значения, если таковые встречаются.

Упражнение

Перепишите программу только что данного примера без использования метода `update()`.

Далее будет рассмотрен более сложный пример употребления словаря — личная телефонная книжка:

Листинг 2.19. Словарь — телефонная книжка

```
# типы телефонов
phone_types = {
```

```
  "h": "домашний",
  "w": "рабочий",
  "m": "мобильный",
}
# содержание записной книжки
phones = {
 "Петя": ("773203", "h", "Петя Петухов", "закадычный друг"),
 "Маша": ("81422835674", "w", "Маша Соколова", "знакомая"),
 "Pekka": ("358401736345", "m", "Pekka Jarvinen", "бизнесмен из Оулу"),
}
# примеры использования записной книжки
print phones["Маша"][2], phones["Маша"][0]
print phones["Pekka"][0], phone_types[phones["Pekka"][1]]
print phones["Петя"][3]
```

Результат выполнения:

```
Маша Соколова 81422835674
358401736345 мобильный
закадычный друг
```

В этом примере ключом служит некоторое короткое прозвище, которое позволяет легче запомнить человека. Каждому ключу соответствует кортеж с информацией о человеке: номер телефона, тип телефона (расшифровывается с помощью дополнительного словаря), имя и фамилия, примечание.

> ***Упражнение***
>
> Приведенная структура данных — не самая удобная и гибкая. Измените структуру данных так, чтобы для любого человека можно было записывать несколько телефонов.

Наверное, не лишним будет заметить, что в Python можно разбивать длинную строку на более короткие внутри всех трех видов скобок: `()`, `[]`, `{}`, что позволяет делать программы более красивыми и удобочитаемыми.

2.13. Осваиваем форматированный вывод

В самом конце листинга 2.15 мы рассмотрели пример, который формировал набор слов по заданным приставкам, корням и суффиксам с окончаниями. Попробуем переделать нашу программу так, чтобы, во-первых, можно было бы задавать число столбцов, и, во-вторых, выровнять столбцы. Первого можно добиться, разделив порождение и вывод слов. Порождаемые слова будем записывать в список, а затем выводить в отдельном цикле. Второго

можно добиться, если применить операцию форматирования с соответствующей строкой формата. Итак:

```
ps = ["при", "за", "от"]
ks = ["корм", "лож", "точ"]
ss = ["", "енный", "ка"]
cols = 4                                # задаем число столбцов
words = []                              # в начале список пуст
for p in ps:
    for k in ks:
        for s in ss:
            words.append(p+k+s)         # добавляем слово к списку
for i in range(len(words)):             # для i от 0 до кол-ва слов без 1
    print "%-15s" % words[i],           # выравнивание по левому краю
                                        # в рамках 15 позиций
    if (i+1) % cols == 0:               # нужна ли новая строка?
        print                           # перевод строки
```

Результат выполнения:

```
прикорм         прикорменный    прикормка       прилож
приложенный     приложка        приточ          приточенный
приточка        закорм          закорменный     закормка
залож           заложенный      заложка         заточ
заточенный      заточка         откорм          откорменный
откормка        отлож           отложенный      отложка
отточ           отточенный      отточка
```

В последнем примере применен метод добавления элемента к списку: `words.append()`. Элемент добавляется к концу списка. В цикле `for i` мы видим два значения операции `"%"`. Во-первых, применение строки формата к аргументам. В данном случае строка формата указывает, что мы выводим строку из 15 символов, выровненную по левому краю. Во-вторых, `"%"` — операция по получению остатка от деления одного числа на другое. Перевод строки требуется каждые `cols` раз — как раз тогда, когда `i+1` делится нацело на `cols`.

Операция форматирования заменяет в строке форматирования (в примере выше это — `"%-15s"`) все записи, начинающиеся с процента и оканчивающиеся буквой, на данные, указанные справа от операции, по очереди. Эта операция позволяет выводить текст с включениями чисел, строк и других данных в поля определенной ширины, выровненные по левому или правому краю. Для чисел строка форматирования дает возможность указывать количество цифр после запятой.

В следующем шуточном примере видны почти все виды возможных форматов:

Листинг 2.20. Зоопарк форматов

```
# Данные, которые будут сидеть в клетках
a = 1
b = -3
c = 11.123966
d = "просто строка"
print """=%40s=
           +------+ %% - процент
    *******|%6i|  ==============
    *%05i*+------+  =%s=
    *******/////// ==============
           /%-6i/    ~~~~~~~~~~~
           ///////    ~~~%5.2f~~~
                      ~~~~~~~~~~~
%s
""" % ("Зоопарк форматов".center(40), b, a, d, b, c, "-"*42)
```

Результат выполнения:

```
=            Зоопарк форматов            =
           +------+ % - процент
    *******|    -3|  ==============
    *00001*+------+  =просто строка=
    *******/////// ==============
           /-3    /    ~~~~~~~~~~~
           ///////    ~~~11.12~~~
                      ~~~~~~~~~~~
------------------------------------------
```

Рассмотрим наших зверей поближе. Первый зверь — строка `"Зоопарк форматов"` — сначала устанавливается по центру полосы из сорока символов (`.center(40)`), а затем встает на место, обозначенное `%40s`. Дальше в строке формата мы встречаем двойной повтор знака `"%"`. Поодиночке проценты стоять в строке формата не могут, т. к. они обозначают места, куда подставляются значения, поэтому для получения собственно процента приходится указывать его дважды. Следующий зверь — значение `b` — содержится в клетке `%6i`. Число `6` говорит о том, что под него выделено 6 символов, а `i` указывает на тип нашего зверя — целое число. Мы видим, что зверь забился в правый угол. Клетка следующего зверя описывается форматом `05i`. Это тоже целое число, на него выделено пять символов, а ноль впереди позволяет зверю равномерно

заполнить клетку незначащими нулями. Следующий представитель форматной фауны — просто строка. Она выводится без всяких прикрас и занимает столько места, какой она длины. Далее мы опять встречаемся с `b`, но теперь уже значение занимает левый край — так получается, когда формат содержит `"-"`. Далее следует более сложный зверь — число с плавающей точкой. Формат для бассейна этого "водоплавающего" зверя тоже непростой: `%5.2f`. Здесь `5` говорит о том, что под число выделено пять знакомест, а после запятой следует указать всего `2` символа. Буква `f` же как раз и говорит, что мы имеем дело с числом с плавающей точкой, т. е., дробным числом. Наконец, `"-"*42` строит ограду нашего зоопарка из 42 звеньев.

В приведенном выше примере есть один недостаток — в строке формата не видно табличек с названиями зверей. Следующий пример исправляет этот недостаток:

Листинг 2.21. Зоопарк форматов (дубль 2)

```
a = 1
b = -3
c = 11.123966
d = "просто строка"
title = "Зоопарк форматов".center(40)
fence = "-"*42
print """=%(title)40s=
           +------+ %% - процент
    *******|%(b)6i|  ===============
    *%(a)05i*+------+  =%(d)s=
    *******////////  ===============
           /%(b)-6i/     ~~~~~~~~~~~
           ////////     ~~~%(c)5.2f~~~
                        ~~~~~~~~~~~
%(fence)s
""" % vars()
```

Функция `vars()` дает нам словарь со значениями переменных, а операция форматирования `"%"` умеет расставить зверей по данным в скобках после знаков процента табличкам. Можете проверить — это тот же самый зоопарк!

Упражнение

Попробуйте написать программу, выводящую поясняющую таблицу с примерами форматов вида:

```
----------+------------
   формат | результат
```

```
----------+-------------
       %s | "строка"
      %5i | "  123"
...
```

2.14. Изучаем работу с файлами

До сих пор мы не могли сделать ничего серьезного из-за того, что наши программы не работали с файлами. *Файл* — не что иное, как последовательность байтов, хранящаяся на носителе данных и доступная по имени. Данные из файла можно читать, в файлы можно записывать, но перед этим файл нужно открыть (open), а после использования — закрыть (close). Еще одна особенность файла состоит в том, что если мы прочитали из файла некоторые данные, снова мы можем их прочитать, только передвинув специальный указатель позиции в файле, поэтому нужно стараться обработать файл за одно прочтение. *(О том, как работать с указателем позиции в файле, можно прочитать в разд. 8.2.9, где описаны функции* `seek()` *и* `tell()`*.)*

Ниже мы рассмотрим пример, в котором порождаемые программой данные записываются в файл.

Листинг 2.22. Простой пример записи информации в файл

```
f = open("file.dat", "wt")        # открываем файл, f будет
                                  # использоваться для работы
                                  # с файлом
for x in xrange(100):             # для x от 0 до 99:
  f.write("%i %i %i\n" % (x, x**2, x**3))  # вывод в файл f значений x,
                                    # их квадратов и кубов
f.close()                         # закрываем файл f
```

В данном примере файл открывается для записи (о чем говорит символ `w`) в текстовом режиме (символ `t`). Теперь мы можем прочитать значения из файла и просуммировать числа:

Листинг 2.23. Читаем из файла

```
import string
f = open("file.dat", "rt")        # открываем файл для чтения
s1 = s2 = s3 = 0                  # обнуляем переменные для
                                  # накопления сумм
while 1:
  stroka = f.readline()           # читаем строку
```

```
  if stroka == "":                # если конец файла,
    break                         # покидаем цикл while
  s = string.split(stroka)        # разбиваем строку на составляющие
  s1 = s1 + int(s[0])             # каждую часть переводим в число
  s2 = s2 + int(s[1])             # и прибавляем к соответствующему
  s3 = s3 + int(s[2])             # счетчику
f.close()                         # закрываем файл f
print s1, s2, s3
```

Вот что получается:

```
4950 328350 24502500
```

Теперь немного другой пример. Пусть в файле mytext.txt записан некоторый текст, и в этом тексте признак конца строки стоит не в конце абзаца, как принято в текстовых процессорах, а в конце каждой строки, чтобы та не была больше 80 символов. Но и это еще не все. В тексте присутствуют знаки переноса в виде дефисов, что, конечно, добавит хлопот при вставке текста в текстовый процессор. К счастью, абзацы отделены дополнительной пустой строкой, что позволяет однозначно определить их границы. Требуется привести (переформатировать) файл к виду, когда признак конца строки обозначает конец абзаца, а переносы не употребляются.

Наверное, для поставленной задачи нет необходимости накапливать абзац и лишь затем выводить его, поэтому алгоритм обработки файла может быть следующим:

```
для каждой строки файла:
  если строка состоит лишь из признака конца строки:
    вывести признак конца строки
  иначе:
    если строка оканчивается на дефис и признак конца строки:
      вывести строку без дефиса и признака конца строки
    иначе:
      вывести строку с пробелом на месте признака конца строки
вывести признак конца строки
```

Теперь процедура обработки файла ясна, и можно составить соответствующую программу:

Листинг 2.24. Программа переформатирования текстового файла

```
inpfile = open("mytext.text", "rt")    # откуда читать
outfile = open("mytext1.text", "wt")   # куда писать
```

```
for l in inpfile.readlines():      # цикл по строкам файла:
  if l[0] == "\n":                 # если пустая строка:
    outfile.write("\n")
  elif l[-2:] == "-\n":            # если на конце дефис:
    outfile.write(l[:-2])
  else:                            # в другом случае:
    outfile.write(l[:-1]+" ")
outfile.write("\n")
inpfile.close()
outfile.close()
```

Листинг 2.25. Файл mytext.txt. Исходный файл

```
В этом абзаце нет ничего
особенного, даже переносов.

А этот абзац будет немно-
го сложнее. Но все-
таки и его можно обрабо-
тать.
```

Листинг 2.26. Файл mytext1.txt. Результат обработки

```
В этом абзаце нет ничего особенного, даже переносов.
А этот абзац будет немного сложнее. Но всетаки и его можно обработать.
```

Внимательный читатель заметит, что дефис в конце строки может быть не только переносом, но и, например, частью тире или собственно дефисом. Исключить первое предлагается читателю в качестве упражнения. Второе, пожалуй, без дополнительной информации исключить достаточно сложно. Можно, к примеру, договориться, что переносы обозначаются другим символом.

Упражнения

Исправьте программу так, чтобы она корректно обрабатывала тире в конце строк. (Под тире можно понимать один или два дефиса, отделенных от слов пробелами.)

Составьте программу для переформатирования текстового файла, где вместо пустых строк для определения абзаца использовать бы отступы красной строки.

2.15. Обработка ошибок

Как мы уже видели, нормальную работу программы могут прервать ошибки. Например, файл, который мы хотели открыть для чтения, может не существовать на диске, а операция деления не знает, как разделить на ноль. Во многих языках программирования, в том числе и Python, для обработки таких исключительных ситуаций имеются специальные средства. Ведь если исключение не обработать, программа просто прервется!

Для перехватывания и обработки исключительных ситуаций служит оператор `try-except`. Английское *to try* означает попытаться, а *to except* — исключить. Синтаксис оператора следующий:

```
try:
  фрагмент программы
except тип исключения1:
  обработчик исключения1
except тип исключения2:
  обработчик исключения2
...
except:
  обработчик всех остальных исключений
```

В приведенном ниже примере мы попробуем ловить исключения, возникающие при вычислениях:

```
# для всех i от -5 до 5:
for i in range(-5, 6):
  try:  # попытаться:
    print i, 1.0/i
  except ZeroDivisionError:  # в случае деления на нуль:
    print "ошибка"
```

Результат выполнения:

```
-5 -0.2
-4 -0.25
-3 -0.333333333333
-2 -0.5
-1 -1.0
0 ошибка
1 1.0
2 0.5
3 0.333333333333
4 0.25
5 0.2
```

В приведенном примере мы ловим и обрабатываем конкретное исключение `ZeroDivisionError`. В одном операторе `try-except` может быть несколько частей `except` для разных исключений. При возникновении исключения работа программы прерывается, и управление переходит к соответствующей части `except`. Если программист не позаботился обработать исключение, Python это сделает сам, прервав программу совсем.

Попробуем переделать пример из предыдущего раздела (где мы суммировали числа из файла) так, чтобы он обрабатывал возникающие исключения:

Листинг 2.27. Читаем из файла

```
import string, sys
try:  # пытаемся:
  f = open("file.dat", "rt")       # открываем файл для чтения
except IOError:  # исключение: ошибка ввода-вывода
  print "Не могу открыть файл file.dat"
  sys.exit(0)  # выход из программы

s1 = s2 = s3 = 0                   # обнуляем переменные для
                                   # накопления сумм
while 1:
  stroka = f.readline()            # читаем строку
  if stroka == "":                 # если конец файла,
    break                          # покидаем цикл while
  s = string.split(stroka)         # разбиваем строку на составляющие
  try:  # пытаемся приплюсовать новые числа
    s1, s2, s3 = s1 + int(s[0]), s2 + int(s[1]), s3 + int(s[2])
  except ValueError:  # исключение из функции int
    print "Встретилось нечисло"
  except IndexError:  # проблемы с индексированием списка s
    print "Не хватает числа (нужно 3 числа в строке)"

f.close()                          # закрываем файл f
print s1, s2, s3
```

Теперь мы отслеживаем ситуации, когда не найден файл, который мы собирались читать, а также случаи, когда в файле оказались не совсем те данные, которые мы ожидали. Наверное, вы заметили, что теперь присваивание значений всем трем переменным происходит как бы параллельно. Если бы мы оставили предыдущий вариант записи, могло бы легко оказаться, что сум-

мирование было бы не совсем корректным. В новом варианте ошибка, возникающая в правой части присваивания, прерывает все присваивание всем трем переменным. Поэтому можно смело сказать, что строка с ошибкой не обрабатывается вообще.

Исключения можно вызывать и самим. Для этого служит оператор `raise`. Например, если мы хотим, чтобы строки содержали точно три числа, мы можем переписать фрагмент программы следующим образом:

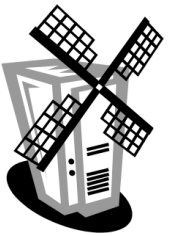
```
while 1:
  stroka = f.readline()          # читаем строку
  if stroka == "":               # если конец файла,
    break                        # покидаем цикл while
  s = string.split(stroka)       # разбиваем строку на составляющие
  try:  # пытаемся приплюсовать новые числа
    if len(s) != 3:
      raise IndexError
    s1, s2, s3 = s1 + int(s[0]), s2 + int(s[1]), s3 + int(s[2])
  except ValueError:  # исключение из функции int
    print "Встретилось нечисло"
  except IndexError:
    print "Ошибка. Нужно точно 3 числа в строке"
```

Python имеет достаточно большой набор встроенных исключений *(см. разд. 8.6.2)* на все исключительные случаи.

Упражнение

Выясните, какое исключение возбуждается при выполнении операции возведения больших чисел в квадрат или куб и перепишите первый пример из предыдущего раздела, обрабатывая это исключение.

Глава 3



Функциональное программирование

Python имеет все необходимое для *функционального программирования*. Ниже мы познакомимся с этим незаслуженно забываемым учебниками подходом.

В предыдущей главе мы рассматривали *процедурное программирование*. Мы говорили интерпретатору Python, как надо решать ту или иную задачу, записывая необходимые для этого действия. Здесь же мы будем, в основном, использовать выражения, указывая интерпретатору, что ему необходимо делать. Выражения объединяют функции между собой в более сложные структуры. Такой метод позволяет работать над программой как будто с помощью конструктора: необходимая схема обработки данных получается тем или иным соединением входов и выходов черных ящиков (наших функций и операций). Содержимое же черных ящиков раскрывается в определениях функций.

Читателям, знакомым с процедурными языками (вроде Basic, Pascal, C или Java), рекомендуется почитать эту главу с особой тщательностью, т. к. функциональный подход к программированию может понравиться далеко не сразу. Тем не менее, он дает программисту очень мощные средства, позволяя писать не только менее раздутые, но и более устойчивые к ошибкам программы. Конечно, совсем необязательно писать чисто функциональные программы. Главное — видеть, где можно применить то или иное средство.

3.1. Чистые функции и их композиция

В математике функция есть нечто, отображающее объекты из одного множества (множества определения функции) в другое (множество значений функции). Такие функции, назовем их *чистыми*, работают "механически", однозначно вычисляя выходное значение по заданным входным. Чистые функции не должны хранить в себе какие-либо данные между двумя вызовами. Их иногда полезно представлять себе черными ящиками, о которых известно только что они делают, но не известно как. На рис. 3.1 приведен пример чистой функции, вычисляющей максимум из двух чисел. (Попутно заметим, что имеющиеся в Python встроенные функции `min()` и `max()` могут работать с произвольным числом аргументов.)

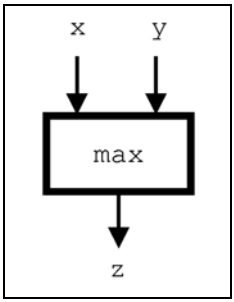


Рис. 3.1. Функция `max()` как черный ящик

Программы в функциональном стиле конструируются как *композиция* определенных ранее функций. На рис. 3.2 мы видим композицию из функций `min()`, `max()` и операции "+". С помощью этой функции можно вычислять сумму минимума и максимума трех чисел.

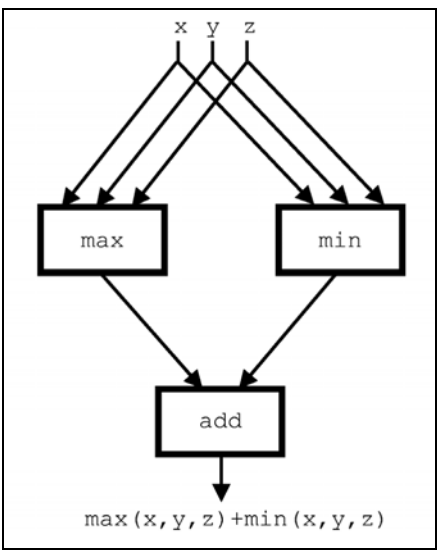


Рис. 3.2. Схема вычисления суммы минимума и максимума трех чисел

Упражнение

Нарисуйте схему функции суммы минимума и максимума трех чисел, если считать, что функции `min()` и `max()` принимают не более двух аргументов.

В *разд. 2.3* мы уже научились определять и применять функции, однако рассмотрим это еще раз, более подробно.

При определении функции с помощью оператора `def` мы даем ей и ее формальным аргументам имена. В теле определения функции мы используем эти имена в вычисляемых выражениях. Результат функции возвращается с помощью оператора `return`, например:

```
def my_fun(x, y): return (x+y)**2
```

Здесь `my_fun` — имя функции, `x` и `y` — ее формальные аргументы.

Есть и другой способ определения функции, с помощью лямбда-выражения:

```
my_fun = lambda x, y: (x+y)**2
```

(В последнем случае мы могли бы и не давать функции имя, а использовать ее прямо в месте создания.)

Функции, разумеется, определяются не для красоты, а с целью применения в программе. Применяя функцию, мы должны дать ей на вход конкретные аргументы (их называют фактическими аргументами):

```
my_fun(1.2, 3.0)
```

В этом примере функция возвратит результат: `17.64`.

Функция `my_fun()` является чистой, т. к. она не делает ничего другого, кроме того, что однозначно вычисляет результат на основе входных данных. На вход мы можем подавать различные данные, неизменно получая результат, вычисляемый по формуле `(x+y)**2`. (Разумеется, если не возникнет ситуации переполнения.)

Теперь мы можем забыть о внутреннем устройстве функции `my_fun()`, помня только, что она делает, и считать ее еще одним доступным для конструирования программы черным ящиком.

> ***Упражнение***
>
> Определите функцию `my_fun2()`, которая будет вычислять значение по формуле `my_fun(a+b, b)+my_fun(a, a-b)`.

Наверное, можно легко догадаться, что операции в выражениях на языке Python тоже являются функциями. Просто они столь часто применяются, что для них придумана специальная запись (например, `+` для сложения или `**` для возведения в степень). Следующий пример показывает, что мы можем записывать операции в виде функций. Для этого нам нужно использовать модуль `operator`, в котором встроенные операции доступны в виде функций:

```
from operator import add, pow
def my_fun(x, y): return pow(add(x, y), 2)
print my_fun(1.2, 3.0)
```

Здесь функция `add()` выполняет сложение двух аргументов, а `pow()` возводит первый аргумент в степень, указанную вторым аргументом. Посмотреть соответствие операций и функций можно в *разд. 9.2.6, в табл. 9.5.*

Упражнение

Запишите с помощью функций из модуля `operator` следующее выражение: `(d/((a+b)*c))-1`.

Функции можно строить не только для чисел, но и для других объектов, например, строк. Многие полезные функции для работы со строками мы уже изучили в предыдущей главе и знаем, что они берутся из модулей `string` и `re`. Например, мы можем определить функцию, которая ставит точку после каждой буквы строки:

```
from string import *
def add_dots(s): return join(s, ".")+"."
print add_dots("PYTHON"), add_dots("IS"), add_dots("SMART")
```

Результат выполнения:

```
P.Y.T.H.O.N. I.S. S.M.A.R.T.
```

Упражнение

Определите функцию `ul()` для расстановки подчеркиваний "_" вокруг символов строки. То есть выражение `ul("PYTHON")` должно давать `"_P_Y_T_H_O_N_"`.

Примечание

В программировании "чистые" функции являются не всегда достигаемым идеалом, т. к., помимо получения результата на основании аргументов, они могут вызывать побочные эффекты в некоторых других частях программы. Более того, некоторые функции и существуют ради таких побочных эффектов (как, например, функции для ввода или вывода)!

Функции в языке Python имеют и еще одну интересную возможность — необязательные аргументы. При описании таких аргументов указываются значения, которые будут использоваться, когда соответствующий аргумент не задан. Говорят, что аргументу задается *значение по умолчанию.* Необязательные аргументы должны следовать после аргументов, не имеющих значений по умолчанию. Следующий пример наглядно иллюстрирует применение необязательных аргументов:

```
def two_words(word1, word2, separator=" "):
  return word1 + separator + word2
```

```
print two_words("это", "слова")
print two_words("раз", "два", "-")
```

Результат выполнения:

```
это слова
раз-два
```

В этом примере функция `two_words()` служит для записи двух слов друг за другом, причем разделитель между этими двумя словами можно задавать третьим аргументом. По умолчанию разделителем является символ пробела.

Упражнение

Придумайте примеры трех-четырех функций, в которых естественным образом можно использовать аргументы по умолчанию.

3.2. Обработка последовательностей

Мы уже научились строить программы для обработки одиночных значений, однако этого почти всегда недостаточно. Часто требуется "массовая" обработка значений из некоторого списка. В Python для этого имеются специальные функции, которые мы и рассмотрим ниже.

Применить одну и ту же функцию ко всем элементам списка можно с помощью функции `map()`. Эта функция принимает в качестве первого аргумента функцию (она и будет применяться ко всем элементам списка или списков). Все остальные аргументы функции `map()` — списки. Число списков должно соответствовать числу аргументов функции.

В следующем примере функция `x2()` применена к каждому элементу списка. Она возвращает удвоенное входное значение. В результате работы функции `map()` получается новый список, в котором все значения в два раза больше, чем в исходном списке:

```
>>> def x2(x):
...   return x*2
...
>>> print map(x2, [1, 2, 3])
[2, 4, 6]
```

Для использования `map()` совсем необязательно, чтобы применяемая функция имела имя. Вот где находят применение лямбда-выражения, позволяющие получить и использовать результат вычисления функции без присвоения ей имени:

```
>>> print map(lambda x: x*2, [1, 2, 3])
[2, 4, 6]
```

Функция `map()` может работать сразу с несколькими списками, как в следующем примере, где склеиваются буквы из трех строк:

```
>>> print map(lambda x, y, z: x+y+z, "123", "abc", "ABC")
['1aA', '2bB', '3cC']
```

Здесь мы увидели пример того, что `map()` считает строку списком букв. Дело в том, что с точки зрения Python и списки, и кортежи, и строки являются последовательностями, с которыми умеет работать функция `map()`.

Упражнение

Напишите функцию, которая вычисляет коды символов из заданной строки. Указание: код символа дает встроенная функция `ord()`.

Не нужно забывать, что элементами списка могут быть последовательности. В следующем примере мы получаем первые буквы слов, разделенных пробелами:

```
from string import split, join
def abbr(s):
  return join(map(lambda x: x[0], split(s)), "")
print abbr("""Python Enhancement Proposal""")
print abbr("""Internet Protocol""")
print abbr("""Simple Mail Transfer Protocol""")
```

Функция `abbr()` действует следующим образом. Сначала функция `split()` разбивает строку на слова. Затем `map()` применяет к каждому слову функцию, определенную с помощью лямбда-выражения `lambda x: x[0]`. То есть от каждой строки списка берется первый символ. И, наконец, функция `join()` объединяет список букв в одну строку.

Разумеется, если у нас целый список фраз, подлежащих сокращению, мы можем применить функцию `map()` дважды:

```
from string import split, join
phrases = ["""Python Enhancement Proposal""",
           """Internet Protocol""",
           """Simple Mail Transfer Protocol""", ]


def abbr(s):
  return join(map(lambda x: x[0], split(s)), "")
print join(map(abbr, phrases), "\n")
```

Результат будет один и тот же:

```
PEP
IP
SMTP
```

Здесь мы применяем функцию `abbr()` к каждому элементу списка `phrases` и объединяем полученные строки, ставя между ними признаки концов строк `"\n"` с помощью все той же функции `join()`.

Упражнение

Напишите программу, которая заменяет каждое слово на звездочку.

Когда требуется выбрать элементы последовательности по определенному критерию, нам поможет функция `filter()`. В следующем примере мы отбираем только положительные числа:

```
>>> print filter(lambda x: x > 0, [1, -2, 0, -3, 6, 10])
[1, 6, 10]
```

А в этом примере из строки удаляются знаки пунктуации:

```
>>> print filter(lambda c: c not in ",.!:?;-()",
...              "Строка, в которой дан (этот) пример.")
Строка в которой дан этот пример
```

Стоит заметить, что функция `filter()` не превращает строку в список, как это делает функция `map()`.

Упражнение

Напишите выражение, с помощью которого можно удалить из списка все четные целые числа.

Еще одна функция для обработки последовательностей — `reduce()` — помогает организовать цепочку операций с участием элементов последовательности. Первый аргумент этой функции — функция. Второй — последовательность, а третий — значение, которое возвращается в случае пустой последовательности (оно участвует в цепочке операций в качетсве самого первого аргумента). Следующий пример демонстрирует получение произведения элементов списка:

```
>>> print reduce(lambda x, y: x*y, [3, 4, 5, 6], 1)
360
```

В этом примере действия были произведены в следующем порядке: `(((1*3)*4)*5)*6`.

Разумеется, при выполнении операций по цепочке можно накапливать значение в списке. В следующем примере строится список, порядок элементов в котором — обратный исходному:

```
>>> def rev(l):
...   return reduce(lambda x, y: [y] + x, l, [])
...
```

```
>>> print rev([1, 2, 3, 4])
[4, 3, 2, 1]
```

Здесь выполнялись следующие действия: `([4]+([3]+([2]+([1]+[]))))`. Конечно, при программировании на Python лучше воспользоваться встроенным методом `reverse()`, *см. статью "Списки" в разд. 8.2.3 табл. 8.5.*

> ***Упражнение***
>
> Используя `reduce()`, проверить, все ли значения в списке, содержащем числа, равны единице.

Функции `map()`, `filter()` и `reduce()` соответствуют наиболее распространенным схемам обработки последовательностей.

3.3. Рекурсия

Внутри функции можно вызывать другие функции, но ничто не мешает функции вызывать саму себя. Функция, которая вызывает себя напрямую или через другие функции, называется *рекурсивной*. У такой функции, чтобы она не "зациклилась", есть как минимум две ветви вычислений: рекурсивная, в которой функция вызывает себя, и нерекурсивная, где дальнейшей рекурсии не происходит, а результат вычисляется другими методами.

Факториал от числа `n` — это произведение всех целых чисел от 1 до `n`, факториал нуля равен единице. Его можно вычислить так:

```
reduce(lambda x, y: x*y, xrange(1, n+1), 1L)
```

> **Примечание**
>
> Здесь мы добавили `L` к записи числа 1 для того, чтобы вычисления производились с произвольной точностью. Если этого не сделать, переполнение произойдет уже при `n`, равном 13.

Авторы учебников программирования очень любят приводить второй пример рекурсивного вычисления факториала:

```
def factorial(n):
  if n < 2:
    return 1
  else:
    return n*factorial(n-1)
```

В первом случае мы действуем прямо по определению: с помощью функции `xrange()` создается список целых чисел от 1 до `n` включительно, а затем функция `reduce()` организует вычисление произведения этих чисел через

цикл. Во втором примере мы вычисляем факториал рекурсивно: через факториал числа, меньшего на единицу, а для "маленьких" чисел 0 и 1 сразу даем ответ. Очевидно, что наша рекурсия гарантированно остановится, ведь при каждом новом вызове `n` уменьшается на единицу. Таким образом, в некоторых случаях циклический алгоритм можно заменить рекурсией, и наоборот.

Упражнение

Напишите рекурсивную функцию для нахождения суммы элементов списка.

Следующий пример демонстрирует применение рекурсии для поиска выхода из лабиринта. Лабиринт разбит на квадратные клетки, некоторые из которых заняты стенами. Проходить из клетки в клетку можно только, если свободные от стен клетки имеют общую сторону.

Листинг 3.1. Поиск пути выхода из лабиринта

```
maze = ["***************",
        "* * *   *     *",
        "*   * * * * * *",
        "* * * *   * ***",
        "* **   ****   *",
        "*    *   *  * *",
        "***************",]

x0, y0 = 13, 5  # начальная позиция
x1, y1 = 3, 1   # координаты выхода

def find_exit_path(maze, x, y, visited):
  """ find_exit_path(maze, x, y, visited)

  найти путь из точки (x, y) лабиринта maze;
  visited содержит список уже посещенных точек.
  """
  if (x, y) == (x1, y1):
    return [(x, y)]  # выход найден
  elif maze[y][x] == "*" or (x, y) in visited:
    return []        # стенка или уже были здесь - не выход
  else:
    # пробуем пройти на все соседние клетки:
    for (dx, dy) in [(1, 0), (0, 1), (-1, 0), (0, -1)]:
```

```
      # ищем путь с соседней клетки
      path = find_exit_path(maze, x+dx, y+dy, visited+[(x, y)])
      if path:       # если путь найден,
        return [(x, y)] + path  # дополняем и возвращаем его
    return []               # ничего не нашли...

print find_exit_path(maze, x0, y0, [])
```

Заметьте, что в этом примере мы отошли от чисто функционального подхода. Во-первых, кроме выражений мы применяли и другие операторы (`if`, `for` и т. д.). Во-вторых, наша функция `find_exit_path()` использует для своей работы координаты выхода `x1` и `y1`, находящиеся вне определения функции. Конечно, второе легко исправить, если передавать `x1` и `y1` как пятый и шестой параметр.

Упражнение

Перепишите функцию `find_exit_path()` для случая, когда соседними считаются клетки с общими углами.

Рекурсия — очень мощный описательный механизм, но не стоит им злоупотреблять. Следующий пример проиллюстрирует, что слепо примененная рекурсия может быть очень неэффективна:

Листинг 3.2. Пример, в котором рекурсия очень неэффективна

```
"""
Числа Фибоначчи образуются по простому правилу:
fib(0) = 0
fib(1) = 1
fib(2) = 0 + 1
...
fib(n) = fib(n-2) + fib(n-1)
"""
def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-2) + fib(n-1)

print fib(3)
```

Однако вычисление даже сотого числа Фибоначчи заставляет компьютер серьезно задуматься, тогда как для следующего решения, использующего обычный цикл, и десять тысяч итераций — не проблема:

```
def fib(n):
  if n < 2:
    return n
  else:
    (a, b) = (0L, 1)
    for i in xrange(n-1):
      (a, b) = (b, a+b)
    return b
print fib(5)
```

В этом примере мы держим только два элемента последовательности чисел Фибоначчи: предпоследний и последний `(a, b)`. Внутри цикла происходит переход к следующим двум последним элементам, который происходит `n-1` раз, после чего `b` содержит искомый элемент.

Ниже мы рассмотрим еще один пример рекурсивной функции — функции для получения всех размещений элементов списка по `M` ящикам. Эта функция нам потребуется в *разд. 4.4*.

Листинг 3.3. Функция для получения списка размещений

```
def distributions(set, M):
  """ distributions(set, M) - > dists
  Список размещений элементов списка set по M"""
  if M == 1:  # простой случай для длины 1
    return map(lambda x: [x], set)
  dists = []  # инициализируем список порождаемых размещений
  # перебираем первый элемент размещения
  for i in set:
    set1 = set[:]   # делаем копию set в set1
    set1.remove(i)  # без элемента, равного i
    # на основе set1 строим все размещения меньшей длины
    # и добавляем i в начало каждого размещения.
    # Добавляем полученные размещения к списку dists
    dists = dists + map(lambda D, x=i: [x]+D,
                        distributions(set1, M-1))
  return dists

print distributions([1, 2, 3], 2)
```

Результат выполнения:

```
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

Основная идея функции `distributions()` состоит в том, что мы по очереди берем элементы `set` и присоединяем их к каждому уже готовому размещению `D` меньшей длины. Размещения меньшей длины получаются вызовом `distributions()` для списка `set`, из которого изъят элемент `i`. Разумеется, в случае, когда `M` равно единице, возвращается просто список списков одиночной длины с элементами множества `set`. Таким образом, мы получаем достаточно компактное описание решения задачи получения всех размещений.

Упражнение

Напишите функцию, аналогичную `distributions()`, но выполняющую размещения с повторениями. То есть элемент списка может появляться в размещении более одного раза.

Предупреждение

Увлекшись рекурсивными функциями, можно легко выйти за рамки допустимой в Python глубины рекурсии. Чтобы исправить ситуацию, попробуйте присвоить более высокое значение параметру `setrecursionlimit` из модуля `sys` (этот параметр обычно равен 1000).

3.4. Функции над функциями

Рассмотренные нами функции `map()`, `filter()` и `reduce()` были не совсем обычными функциями: они получали в качестве одного из аргументов функцию. В языке Python функции являются такими же объектами, как строки и числа, и могут передаваться внутрь функции и возвращаться в качестве результата. Внутри функции можно определять другие функции (как с помощью `lambda`, так и оператором `def`). Эти возможности позволяют создавать более гибкие и "умные", и вместе с тем компактные программы.

Первый пример реализует аналог функции `filter()` для словарей:

Листинг 3.4. Функция `dict_filter()`

```
def dict_filter(f, d):
  new_dict = {}
  for (k, v) in d.items():
    new_dict[k] = f(v)
  return new_dict
```

```
def sqr(x):
  return x**2
print dict_filter(sqr, {1: 2, 3: 4})   # возводим элементы в квадрат
```

Результат выполнения:

```
{3: 16, 1: 4}
```

Функция `dict_filter()` преобразует все значения из словаря, создав новый словарь. Преобразование осуществляется с помощью функции одного аргумента `sqr()`, возведение в квадрат, которая передается в `dict_filter()` в качестве первого аргумента.

Функции в Python можно не только передавать внутрь других функций, но и возвращать в качестве результата. Следующий пример создает функцию, которая возвращает функцию, умеющую умножать на заданное число:

Листинг 3.5. Генератор умножителей

```
def gen_mul(n):
  def amul(x, number=n):
    return x*number
  return amul

x2 = gen_mul(2)
x3 = gen_mul(2)
print x2(34), x3(5), x3(10)
print gen_mul(4)(5)
```

Результат выполнения:

```
68 10 30
20
```

В приведенном примере мы с помощью функции `gen_mul()` сначала порождаем функцию, задачей которой является умножение на некоторое фиксированное число.

Необходимо заметить, что в Python 2.0 и ниже число `n` не передать внутрь определения функции иначе, как использованием механизма необязательных аргументов. То есть функция `amul()` на самом деле является функцией двух аргументов: просто второй аргумент можно не задавать — для него будет использовано значение `n`.

В версиях Python 2.1 и выше можно не использовать этот трюк и написать проще:

```
from __future__ import nested_scopes
def gen_mul(n):
  def amul(x):
```

```
    return x*n
  return amul
```

Для версий Python 2.2 и выше `from __future__ ...` указывать не нужно.

Упражнение

Напишите функцию, которая принимает в качестве аргумента две функции (скажем, `f(x)` и `g(x)`) и выводит в качестве результата их суперпозицию `f(g(x))`.

Изученные в этом разделе функции, принимающие в качестве аргументов или возвращающие другие функции в результате, называют *функциями высшего порядка*. В языке Python функции высшего порядка находят свое применение. В следующем примере мы имеем возможность выполнять произвольные действия над найденными в тексте фрагментами. Если вы помните, в *разд. 2.10* был пример, в котором мы заменяли в тексте все числа на звездочки. Оказывается, `re.sub()` своим вторым аргументом может принимать функцию, которой и будет передаваться информация о найденном фрагменте. От этой функции ожидается строка с текстом для замены фрагмента.

Следующий пример поможет найти все целые числа в строке и увеличить их на единицу:

Листинг 3.6. Файл plus1.py. Увеличение чисел в тексте на единицу

```
import re
def add1(m):
  return str(int(m.group())+1)
text = """10, 12; 123 (127)/2"""
text1 = re.sub("[0-9]+", add1, text)
print text1
```

Результат выполнения:

```
11, 13; 124 (128)/3
```

Здесь `m.group()` дает найденный фрагмент целиком, `int()` преобразует его в целое число, а `str()` преобразует число опять в строку. (Пусть запись `m.group` вас не пугает. В *гл. 5* мы узнаем, что это означает. Здесь же нам достаточно знать, что `m.group()` дает нам найденную группу символов.)

Следующий пример носит более практический характер и позволяет производить в тексте *множественные замены* одних подстрок на другие:

Листинг 3.7. Множественные замены подстрок

```
import re
from string import join
```

```
def multireplace(txt, repl_dict):
  def repl_func(match, repl_dict=repl_dict):
    m = match.group(1)
    if repl_dict.has_key(m):
      return repl_dict[m]
    else:
      return ""

  repl_pattern = "(" + join(map(re.escape, repl_dict.keys()), "|") + ")"
  print "Рег.выр.:", repl_pattern
  return re.sub(repl_pattern, repl_func, text)

text = """В этом тексте произведено сразу несколько замен."""

print multireplace(text, {'ко': '..',
                          'о': '_',
                          'ро': '--'})
```

Результат выполнения:

```
Рег.выр.: (\р\о|\о|\к\о)
В эт_м тексте п--изведен_ сразу нес..ль.. замен.
```

Как и в предыдущем примере, для выполнения замен используется функция `re.sub()`. В качестве второго аргумента этой функции передается функция, выполняющая нужные замены на основе переданного ей словаря `repl_dict`. Интересным здесь является применение наших старых знакомых `map()` и `join()` для формирования регулярного выражения (в функции `multireplace()` регулярное выражение выводится оставленным для отладки оператором `print`). Функция `re.escape()` экранирует символы с помощью "\", т. к. иначе они могли бы испортить регулярное выражение и привести к непредсказуемому результату. (Функция `re.escape()` сочла русские буквы опасными для регулярного выражения и поставила перед каждой из них обратную косую черту.)

3.5. Функции могут все

В этой главе мы изучили практически все полезные элементы функционального программирования, присущие языку Python (композиция функций, применение функции к элементам последовательности, рекурсия, функции высшего порядка). Изучая программирование на Python, вы не раз встретитесь с функциями и их очень смелым употреблением во всевозмож-

ных ситуациях, а после некоторой практики научитесь метко находить компактные и эффективные решения задач обработки данных. Совсем необязательно, чтобы решение задачи строилось чисто функциональным путем. Главное, чтобы оно было четким, эффективным и... красивым.

Напоследок читателю предлагается в качестве упражнения разобраться в следующей программе, реализующей алгоритм сортировки списка по возрастанию преобразованием в упорядоченное бинарное дерево и обратно.

Листинг 3.8. Программа сортировки с помощью упорядоченного бинарного дерева

```
def Insert(tree, item):
  """Вставка нового элемента в бинарное дерево"""
  if tree:
    if tree[1] > item:
      return (Insert(tree[0], item), tree[1], tree[2])
    else:
      return (tree[0], tree[1], Insert(tree[2], item))
  else:
    return (None, item, None)

def Flatten(tree):
  """Преобразование дерева в список"""
  if tree:
    return Flatten(tree[0]) + [tree[1]] + Flatten(tree[2])
  else:
    return []

def MakeTree(list):
  """Формирование бинарного дерева из списка последовательным
  включением в него элементов функцией Insert().
  У всех нелистовых узлов дерева имеются левое и правое
  поддеревья, а также значение, присвоенное данной вершине.
  Информация об узле хранится в трехэлементном кортеже.
  Лист - узел, не имеющий поддеревьев, - тоже кортеж
  из трех элементов, но его 0-й и 2-й элементы - None.
  """
  return reduce(Insert, list, None)
```

```
def TreeSort(list):
  """Сортировка с использованием дерева"""
  return Flatten(MakeTree(list))

print TreeSort([1, 5, 3, 2, 12, -1, 2, 1, 4])
```

Конечно, эта чисто учебная программа сортирует гораздо медленнее, чем встроенная в Python функция сортировки. Мы надеемся, что с ее помощью читатель легко разберется в представленном алгоритме сортировки с помощью бинарного дерева и оценит компактность записи функциональной программы.

Глава 4



Строим модули

В этой главе мы будем заниматься более объемными и серьезными программами, чем раньше, научимся разрабатывать структуру программы, правильно разбивать программу на *модули*.

Работая с Python, мы не раз подключали к программе требуемые для решения задачи дополнительные модули. Это были `math`, `string`, `re`, `operator` и т. д. Каждый модуль добавлял новые возможности, предоставляя в наше распоряжение готовые функции и константы.

Метод, при котором программа разбивается на модули, содержащие тематически связанные данные и функции (программные модули), называется *модульным программированием*. Ниже мы рассмотрим его более подробно.

4.1. Что такое модуль

Модульное программирование позволяет разрабатывать и отлаживать программу по частям, сохраняя структуру программы обозримой, а саму программу — удобочитаемой. Конечно, для этого разбиение, или *декомпозиция*, на модули должна быть разумной. Любая более или менее непростая программа имеет множество связей между своими фрагментами: одна функция может вызвать другую, та, в свою очередь, вызвать третью, и т. д. Если в отпечатанном листинге никак не структурированной программы нарисовать все эти вызовы, получится паутина. И чем сложнее программа, тем гуще будет паутина. Однако даже в этой паутине мы сможем заметить, что некоторые группы функций теснее связаны между собой, чем с другими функциями. Есть также группы функций, которые почти не вызывают функции из других групп, но решают общую задачу, и их хотелось бы видеть расположенными вместе. Модуль — это как ящичек с инструментами одного назначения. Есть инструменты слесарные, плотницкие, для работы в саду или в огороде, инструменты для ремонта часов и для ремонта грузовиков... В отличие от плотников или портных, программистам часто требуются уникальные наборы инструментов, которые они разрабатывают к своим программам сами и размещают по ящичкам-модулям.

Все эти наблюдения и аналогии помогают провести разумную декомпозицию программы на модули. Строгих правил для декомпозиции не существует, можно дать лишь некоторые общие рекомендации и привести примеры.

- В виде отдельного модуля обычно оформляется группа функций и данных (как правило, констант), тесно связанных друг с другом. Например, модуль `time` в Python занимается подсчетом времени, преобразованием времени из одного формата в другой и тому подобными манипуляциями со временем.
- Нужно стремиться к тому, чтобы модуль был прост в использовании, а имеющиеся в нем функции — документированы. Например, мы можем понять, как работать с модулем `copy`, дав команду `import copy; print copy.__doc__`. (В документации к модулям обычно приводятся примеры их использования.)
- Желательно, чтобы модуль имел как можно меньше внутренних данных, изменяемых посредством вызова функций или (вообще недопустимо) напрямую. Дело в том, что хранение модулем некоторых изменяемых данных (от которых зависит выполнение предоставляемых модулем функций) может приводить к весьма трудноуловимым ошибкам, а поведение самого модуля может стать непредсказуемым. Разумеется, есть случаи, когда без хранения таких данных не обойтись. Например, модуль `random`, содержащий генератор псевдослучайных чисел, хранит, а при каждом обращении к датчику случайных чисел — изменяет, значение "затравки" (seed), необходимой для вычисления случайного числа.
- Если вы никак не можете построить модуль, который бы не хранил данные о своем внутреннем состоянии, попробуйте после прочтения этой главы изучить и следующую, посвященную объектному методу. Научившись работать с объектами, вы сможете сделать свой модуль, описав в нем класс объектов. Например, тот же модуль `random` (фактически — `whrandom`) для хранения своего внутреннего состояния использует специальный объект.
- Модуль не должен быть очень большим. Если модуль стал слишком велик, рассмотрите возможность его дальнейшей декомпозиции на более мелкие модули. Главное, чтобы разбиение было естественным. Например, модуль `mailbox` для работы с почтовыми ящиками использует другой модуль, упрощающий работу с отдельными письмами. Тот, в свою очередь, использует другие необходимые ему модули.
- Иногда в модуль собирают функции, которые часто используются в рамках одной или нескольких программ. Конечно, такое достаточно случайное собрание функций не делает модуль красивым. Теоретически в этом случае надо бы отвести для каждой функции отдельный модуль. На практике же часто загружаемые, пусть и совсем не связанные между собой,

функции помещают в один модуль. Однако, почему же не связанные? Можно утешить себя мыслью о том, что они связаны между собой тем, что составляют ящичек с инструментами для вашего проекта. Разумеется, как только такой модуль станет слишком тяжелым, инструменты можно распределить по отдельным ящикам.

В языке Python для хранения модуля требуется отдельный файл. Этот файл должен находиться в каталогах, где Python ищет свои модули. Обычно в список таких каталогов входит и каталог, в котором находится исполняющаяся программа. Путь для поиска модулей можно узнать так:

```
>>> import sys
>>> sys.path  # список путей
['', '/usr/lib/python2.1', '/usr/lib/python2.1/plat-linux2',
'/usr/lib/python2.1/lib-tk', '/usr/lib/python2.1/lib-dynload',
'/usr/lib/python2.1/site-packages',
'/usr/lib/python2.1/site-packages/Numeric']
```

Примечание

Во всех примерах мы будем предполагать, что наши модули и программа находятся в одном каталоге.

4.2. Маленький проект

В этом разделе мы осуществим разработку модулей для игровой программы под названием "Быки и коровы", в которой компьютер будет играть с человеком, сам с собой или служить арбитром между двумя игроками.

Любой проект, в том числе и программный, подразумевает стадию *проектирования (дизайна)*. Чем крупнее проект, тем больше времени должно уделяться проектированию. При модульном подходе проектирование обычно заключается в определении набора модулей, составляющих программу, и взаимосвязей между этими модулями. Уже потом, модуль за модулем, мы постепенно детализируем проект.

В хорошо спроектированной программе сразу видно, что модули располагаются на разных уровнях. Самый верхний уровень занимает *главный модуль*, который использует модули более низкого уровня. Те, в свою очередь, могут использовать модули, лежащие еще ниже.

Разные методики рекомендуют различный порядок проектирования модулей. При *нисходящем проектировании* сначала проектируют модули более высокого уровня, а затем опускаются все ниже и ниже. Есть и противоположный подход: сначала проектируются модули нижнего уровня, а затем уже те, что лежат

выше, пока, наконец, не будет достигнут самый верхний уровень. Порядок реализации (программирования) модулей тоже может быть различным.

Заметьте, что во втором случае проще производить отладку программы (ведь для отладки модуля все нижележащие уровни уже присутствуют). На практике мало кто подходит так строго к порядку создания и отладки модулей. Например, для начала можно из нескольких разноуровневых модулей создать некоторый работающий скелет, к которому затем добавляются другие модули, дающие программе требуемую функциональность. При любом подходе для отладки модулей могут потребоваться *заглушки* — модули, которые минимально имитируют соответствующие полнофункциональные модули.

В начале необходимо определиться с *постановкой задачи*. Наша задача — написать программу, которая будет реализовывать некоторую стратегию игры "Быки и коровы" (эта игра известна также под названием "Master mind"). Суть игры состоит в следующем. Каждый из двух играющих загадывает шифр из N десятичных цифр (в классической игре N равно четырем). Цифры в этом шифре не должны повторяться. Затем игроки по очереди делают попытки отгадать код противника, предлагая свой вариант шифра. Противник при этом отгадывает количество "быков" (цифр, совпавших по значению и по месту) и "коров" (цифр, которые имеются в шифре, но стоят не на своих местах). Тот, кто за меньшее число ходов отгадает код (то есть получит ответ "N быков"), побеждает. Ниже приведен пример с угадыванием шифра:

```
4867 - 2 к
2076 - 3 к
7609 - 2 б
7625 - 2 б 1 к
7632 - 4 б
```

Проделав анализ задачи, можно выделить следующие модули (табл. 4.1).

Таблица 4.1. *Назначение модулей проекта игры "Быки и коровы"*

Название	Назначение
BK	Главный модуль, координатор игр
BK_user	Модуль интерфейса с пользователем
BK_strategy	Модуль реализации игровой стратегии (отгадывания шифра)
BK_init	Модуль загадывания шифра
BK_calc	Модуль подсчета быков и коров
Combinatorial	Модуль комбинаторных алгоритмов

Можно предположить, что использование этих модулей выстроится согласно схеме на рис. 4.1. (Здесь не указано, что могут использоваться и другие модули, например, стандартные модули Python).

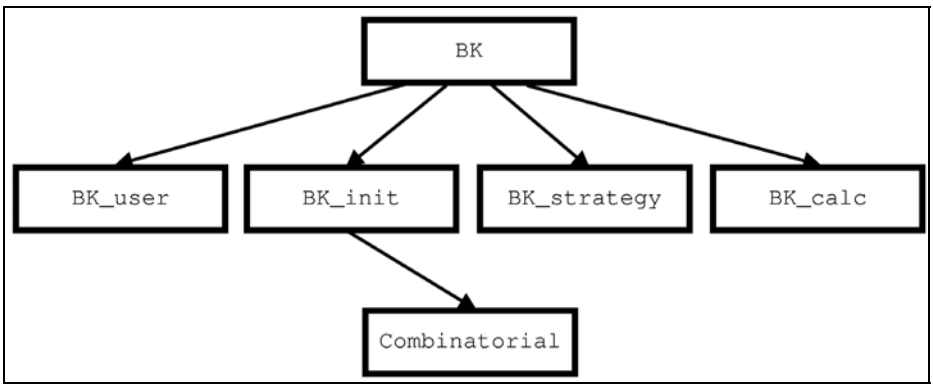


Рис. 4.1. Взаимосвязи модулей проекта игры "Быки и коровы" (вариант 1)

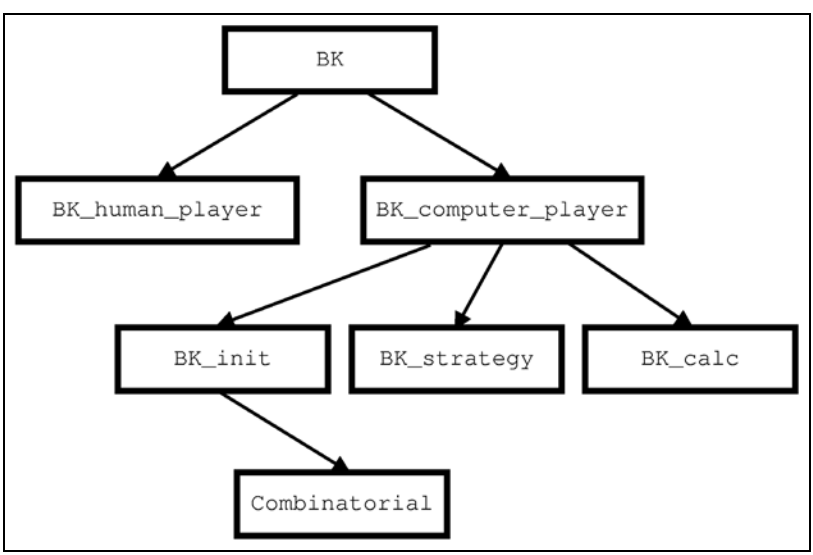


Рис. 4.2. Взаимосвязи модулей проекта игры "Быки и коровы" (вариант 2)

Разумеется, наш модульный дизайн может претерпеть изменения в результате более детальной проработки, но стоит отметить, что наша задача существенно упростилась. Нам осталось определить, что именно и в каком формате модули будут передавать и принимать друг от друга (определить *интер-*

фейс модулей). После этого мы сможем написать собственно программный код модулей.

> ***Упражнение***
>
> Попробуйте составить свой вариант декомпозиции проекта игры "Быки и коровы" на модули.

Если вы выполнили упражнение, то, может быть, пришли к тому, что компьютер и пользователь должны быть уравнены в качестве игроков, т. е. в нашу схему просятся еще два модуля, скажем `BK_human_player` и `BK_computer_player`, чтобы абстрагироваться от того, кто именно играет. Модуль `BK_user` можно убрать. Эти модули должны иметь одинаковый интерфейс, что позволит устраивать матчи между разными игроками. Новый вариант дизайна показан на рис. 4.2.

Соответственно табл. 4.2 представляет описания всех модулей второго варианта.

Таблица 4.2. *Назначение модулей проекта игры "Быки и коровы" (вариант 2)*

Название	Назначение
`BK`	Главный модуль, координатор игр
`BK_human_player`	Модуль для интерфейса с игроком-человеком
`BK_computer_player`	Модуль для интерфейса с игроком-компьютером
`BK_strategy`	Модуль реализации игровой стратегии (отгадывания шифра)
`BK_init`	Модуль загадывания шифра
`BK_calc`	Модуль подсчета быков и коров
`Combinatorial`	Модуль комбинаторных алгоритмов

В получившейся схеме мы можем со временем добавить игроков, которые играют по сети или, например, поручить проверку задуманного человеком шифра компьютеру.

> ***Упражнение***
>
> Попробуйте параллельно обсуждению проекта программы для игры в "Быки и коровы" разрабатывать и реализовывать игру "Морской бой", самостоятельно решив, чем можно воспользоваться из модулей `BK`, а что лучше добавить или переделать.

4.3. Интерфейсы модулей

В предыдущем разделе мы определились, как именно наша программа будет разбита на модули (см. рис. 4.2). Настало время определить интерфейс каждого модуля. Что такое интерфейс функции, мы уже знаем: это набор и тип входных аргументов, возвращаемое значение и т. п. То есть в интерфейс входит все, через что внешний мир "общается" с данной функцией. То же самое верно и для модулей. В интерфейс модуля входят все его функции, предполагаемые к вызову извне, данные, которые можно читать извне модуля.

Начнем с главного модуля — `ВК`. От этого модуля мы ожидаем функцию для организации конкретной игры, поэтому назовем ее `ВК.game()`. В качестве аргументов этой функции нужно передать как параметры самой игры (название, размер шифра), так и некоторые данные об игроках (имена и типы игроков). Название и имена игроков — строки, в качестве размера шифра можно взять одноэлементный кортеж (на вырост — вдруг в дальнейшем будут применяться и другие параметры). Несколько сложнее с типом игрока (таких типов у нас пока два: игрок-человек и игрок-компьютер). К счастью, Python позволяет передавать в качестве аргумента модули, поэтому мы можем передавать функции `game()` собственно модули, реализующие тот или иной тип игрока. Возвращать функция `game()` будет результат поединка.

В любой модуль на Python можно встроить тестовую функцию, демонстрирующую возможности модуля. Назовем ее просто `test()`.

Таким образом, интерфейс модуля `ВК` можно описать так:

Листинг 4.1. Интерфейс модуля ВК

```
def game(game_name, game_format,
         player1_name, player2_name,
         player1_module, player2_module):
  """game(game_name,  -- название игры
         game_format, -- кортеж. 0-й элемент -- размер шифра
         player1_name, player2_name, -- имена игроков
         player1_module, player2_module) -- модули игроков
     -> результат_поединка

  Координация игры двух игроков.
  """

def test():
  """test()
```

```
  Тест модуля BK.py
  """
```

Фактически мы уже стали писать программу, начав, как это положено, с документации. (Наполнением функций кодом мы займемся только в следующем разделе.)

Далее нам нужно написать интерфейсы к модулям `BK_human_player` и `BK_computer_player`. Интересно, что интерфейсы этих модулей должны быть совершенно одинаковыми. В вышележащем модуле `BK` мы абстрагируемся от типа игрока, но нам важно, чтобы игрок правильно отвечал на наши запросы. То есть в модуле должны присутствовать определенные функции с определенными аргументами и регламентированным возвращаемым значением.

Игра "Быки и коровы" — диалоговая игра, поэтому от игрока требуется не только делать ходы, но и отвечать на запросы противника. К счастью, все эти вопросы и ответы строго регламентированы. Давайте мысленно представим, что мы собираемся с кем-то играть в "Быки и коровы". Сначала каждый играющий должен загадать шифр оговоренной длины (назовем соответствующую функцию `start_game()`). В дальнейшем каждый из играющих спрашивает другого о том, сколько быков и коров в шифре-предположении, а другой играющий должен честно ответить. Так как в нашем случае игра происходит при посреднике, нам потребуются две функции: `make_move()` (сделать ход, т. е. выдвинуть свое предположение о шифре противника) и `give_answer()` (ответить на запрос противника). Соответствующая программа может выглядеть так:

Листинг 4.2. Интерфейс модуля `BK_human_player` или `BK_computer_player`

```
def start_game(game_info):
  """start_game(game_info)

  Предложение начать игру, задумать шифр"""

def make_move(game_info):
  """make_move(game_info) -> [c1, c2, ..., cN]

  Предложение сделать ход. Возвращает список цифр
  шифра-предположения
  """

def give_answer(game_info, move_info):
  """give_answer(game_info, move_info) -> (b, k)
```

```
  Предложение ответить на запрос по шифру move_info,
  возвратив кортеж (число быков, число коров)"""
```

Теперь необходимо уточнить, что будет входить в `game_info` (информация об игре с точки зрения игрока) и `move_info` (информация о ходе игры). С `move_info` все более или менее понятно — это шифр-догадка, т. е. список цифр. А вот для `game_info`, пожалуй, потребуется словарь. В табл. 4.3 предложен один из вариантов для `game_info`.

Таблица 4.3. *Ключи и значения словаря* `game_info`

Ключ	Значение
`'game_name'`	Название игры
`'game_format'`	Кортеж, нулевой элемент которого — размер шифра
`'name'`	Имя игрока
`'game_history'`	История ответов соперника
`'player_memory'`	Внутренние данные игрока

Упражнение

Что будет содержать `game_info` в проекте игры "Морской бой"?

Теперь можно заняться интерфейсами модулей `BK_init`, `BK_strategy` и `BK_calc`. Вот они:

Листинг 4.3. Интерфейс модуля `BK_init`

```
def think_of_number(N, M=10):
  """think_of_number(N, M=10) -> code

  Задумать шифр из N цифр"""
```

Листинг 4.4. Интерфейс модуля `BK_strategy`

```
def make_move(game_format, game_history, player_memory):
  """make_move(game_format, game_history, player_memory) -> move_info

  Сделать ход"""
```

Листинг 4.5. Интерфейс модуля `BK_calc`

```
def calc_bk(code1, code2):
  """calc_bk(code1, code2) -> (b, k)

  Подсчет числа быков и коров в одном шифре относительно другого.
  """
```

Здесь уже требуется меньше пояснений. Параметр `N` задает длину шифра, а `M` сейчас равен десяти, но ничто не мешает его изменить. Параметры `game_format`, `game_history` и `player_memory` объяснены в табл. 4.3, а `player_memory` позволяет игроку хранить данные между ходами и задуманный шифр. Разумеется, в функциях модуля `BK_human_player` ключ `player_memory` не используется, т. к. игрок-человек хранит шифр у себя в памяти.

Осталось описать интерфейс модуля `Combinatorial`, который должен содержать функцию `distributions()` для порождения всех размещений элементов некоторого множества по `M`-ячейкам.

Листинг 4.6. Интерфейс модуля `Combinatorial`

```
def distributions(set, M):
  """ distributions(set, M) - > dists
  Список размещений элементов списка set по M"""

def test():
  """тест модуля Combinatorial"""
```

Здесь `set` обозначает список элементов, из которых будут строиться размещения. Параметр `M` обозначает длину порождаемых размещений. Возвращаемое значение должно представлять собой список размещений.

Упражнение

Разработайте интерфейсы для всех модулей проекта игры "Морской бой".

4.4. Разрабатываем модули

Теперь, когда мы определились с набором модулей, их интерфейсами и взаимосвязями, мы можем приступить к разработке модулей. С начала продумаем порядок, в котором мы будем писать модули. Если нам важно во что бы то ни стало быстро показать работающую версию программы, мы можем реализовать модули `BK` и `BK_human_player`. В этом случае у нас будет воз-

можность посмотреть главный модуль в действии и, возможно, увидеть какие-то недостатки в нашем дизайне. Если у нас нет необходимости спешить, можно начать с модуля наиболее низкого уровня `Combinatorial`, отладить его, а затем перейти к модулям более высокого уровня, отладка которых облегчается тем, что все, что нужно для их работы, уже написано. Наконец, можно легко распределить модули между несколькими программистами, делая работу параллельно.

Функция `distributions(set, M)` была разобрана ранее, в *разд. 3.3*. Вот так может выглядеть готовый модуль `Combinatorial`:

Листинг 4.7. Файл Combinatorial.py. Модуль `Combinatorial`

```
def distributions(set, M):
  """ distributions(set, M) - > dists
  Список размещений элементов списка set по M"""
  if M == 1:  # простой случай для длины 1
    return map(lambda x: [x], set)
  dists = []  # инициализируем список порождаемых размещений
  # перебираем первый элемент размещения
  for i in set:
    set1 = set[:]   # делаем копию set в set1
    set1.remove(i)  # без элемента, равного i
    # на основе set1 строим все размещения меньшей длины
    # и добавляем i в начало каждого размещения.
    # Добавляем полученные размещения к списку dists
    dists = dists + map(lambda D, x=i: [x]+D,
                        distributions(set1, M-1))
  return dists

def test():
  print distributions(range(3), 2)

if __name__ == "__main__":
  test()
```

Перейдем теперь к модулю `BK_init`. В этом модуле игрок-компьютер задумывает шифр.

Упражнение

Придумайте алгоритм для загадывания шифра для игры "Быки и коровы". Указание: модуль `random` имеет функцию `choice()`, которая по заданному списку случайно выбирает элемент списка.

Для задумывания шифра длины N есть следующий достаточно простой алгоритм:

```
список цифр = цифры от 0 до 9 включительно
шифр = пустой список
пока длина шифра < N:
  Случайно выбрать цифру из списка цифр
  Поместить выбранную цифру в шифр
  Убрать эту цифру из списка цифр
```

Перевести этот алгоритм на Python не составляет труда:

Листинг 4.8. Файл BK_init.py. Модуль `BK_init`

```
import random
def think_of_number(N, M=10):
  """think_of_number(N, M=10) -> code
  Задумать шифр из N цифр"""
  digit_list = range(M)
  code = []
  while len(code) < N:
    c = random.choice(digit_list)
    code.append(c)
    digit_list.remove(c)
  return code

def test():
  """Тест для модуля BK_init"""
  print think_of_number(4)
```

Следующий модуль — `BK_calc` — тоже достаточно прост, и читатель без труда в нем разберется:

Листинг 4.9. Файл BK_calc.py. Модуль `BK_calc`

```
def calc_bk(code1, code2):
  """calc_bk(code1, code2) -> (b, k)
```

```
  Подсчет числа быков и коров в одном шифре относительно другого.
  """
  b, k = 0, 0
  N = len(code1)
  for i in range(N):
    for j in range(N):
      if code1[i] == code2[j]:
        if i == j:
          b = b + 1
        else:
          k = k + 1
  return b, k
```

А вот модуль `BK_strategy`, пожалуй, потребует бóльших усилий, т. к. необходимо продумать стратегию игры в "Быки и коровы", достаточно четкую для написания алгоритма.

Упражнение

Попробуйте придумать стратегию для игры в "Быки и коровы".

Предлагается следующая стратегия. В самом начале множество возможных ходов — множество всех возможных шифров длины N. То есть мы можем выбрать любой шифр, например, с помощью датчика случайных чисел. Ответ противника уменьшает множество ходов до тех, которые соответствуют указанным противником количествам быков и коров. Определив новое множество возможных ходов, мы опять выбираем любой ход из него. И так далее, пока шифр противника не будет отгадан. Возможно, указанная стратегия не самая лучшая, но она достаточно хороша на практике.

Следующая функция реализует описанную стратегию:

Листинг 4.10. Файл BK_strategy.py. Модуль `BK_strategy`

```
def make_move(game_format, game_history, player_memory):
  """make_move(game_format, game_history, player_memory) -> move


  Сделать ход
  """
  if game_history == []:  # начало игры
    # Создаем список возможных ходов
    player_memory['moves'] = Combinatorial.distributions(
      range(10), game_format[0])
```

```
    # Делаем ход, случайно выбирая один из возможных шифров
    return random.choice(player_memory['moves'])
  else:  # продолжение игры
    # берем информацию о последнем ходе
    last_move_code, last_move_result = game_history[-1]
    # Фильтруем возможные ходы по информации от последнего хода
    new_moves = []
    for code in player_memory['moves']:
      if BK_calc.calc_bk(last_move_code, code) == last_move_result:
        new_moves.append(code)
    # Если больше нет ходов, значит противник давал неверные ответы
    if not new_moves:
      raise ValueError, "Нечестная игра?"
    player_memory['moves'] = new_moves
    # Делаем ход
    return random.choice(player_memory['moves'])
```

Здесь необходимо напомнить, что `game_history` — список двухэлементных кортежей. Первый элемент кортежа содержит список цифр шифра-догадки, а второй — кортеж из двух элементов (`быки`, `коровы`). Поэтому `game_history[-1]` дает последний элемент списка, т. е. результат последнего хода.

В случае, когда список возможных ходов пуст, нужно вызвать сообщение об ошибке. В Python это делается с помощью оператора `raise`. Здесь мы использовали для своих нужд стандартное исключение `ValueError`. *(Об исключениях мы говорили в разд. 2.15.)*

Имея все необходимые модули нижнего уровня, мы можем приступить к модулям `BK_human_player` и `BK_computer_player`. Первый из этих модулей может выглядеть следующим образом:

Листинг 4.11. Файл BK_human_player.py. Модуль `BK_human_player`

```
import string

def _show_code(code):
  """Вспомогательная функция для вывода шифра в виде строки"""
  return string.join(map(str, code), "")

def start_game(game_info):
  """start_game(game_info)
```

```
  Предложение начать игру, задумать шифр"""
  print """%(name)s!
Задумайте шифр для игры '%(game_name)s'.""" % game_info

def make_move(game_info):
  """make_move(game_info) -> [c1, c2, ..., cN]

  Предложение сделать ход. Возвращает список цифр
  шифра-предположения
  """
  print game_info['game_name']
  # выводим историю игры
  for (code, (b, k)) in game_info['game_history']:
    print _show_code(code), " -- ", b, k
  move = raw_input("%(name)s, Ваш ход: " % game_info)
  return map(int, move)

def give_answer(game_info, move_info):
  """give_answer(game_info, move_info) -> (b, k)

  Предложение ответить на запрос по шифру move_info,
  возвратив кортеж (число быков, число коров)"""
  print game_info['name'], "!"
  code = _show_code(move_info)
  b, k = input("Сколько быков, коров в %s? " % code)
  return (b, k)
```

Думается, читатель без труда разберется в этом модуле. Напомним, что `map()` применяет некоторую функцию ко всем элементам списка, `string.join()` сливает список строк в одну строку. Для вывода шифра в виде строки цифр была создана вспомогательная функция `_show_code()`. Эта функция не предназначена для вызова из других модулей (мы ее не включаем в интерфейс модуля) и потому ее имя начинается с подчеркивания. Это как условный сигнал, говорящий о том, что эта функция — внутреннее дело модуля `BK_human_player`.

Конечно, наш модуль не идеален, т. к. он совсем не проверяет значения, которые вводит человек.

Упражнение

Перепишите функции модуля `BK_human_player` так, чтобы проверять ошибки ввода, которые допускает человек. Возможно, для этого придется ввести еще несколько вспомогательных функций.

Следующий модуль, которым мы займемся, — `BK_computer_player`. Собственно, мы уже написали все, что требуется для этого модуля. Остается только соблюсти заданный интерфейс:

Листинг 4.12. Файл BK_computer_player.py Модуль `BK_computer_player`

```
import BK_init, BK_calc, BK_strategy

def start_game(game_info):
  """Предложение начать игру, задумать шифр"""
  # Задумываем код
  secret_code = BK_init.think_of_number(game_info['game_format'][0])
  # Записываем его в словарь память игрока (player_memory)
  game_info['player_memory'] = {'code': secret_code}

def make_move(game_info):
  """Предложение сделать ход"""
  return BK_strategy.make_move(game_info['game_format'],
            game_info['game_history'],
            game_info['player_memory'])

def give_answer(game_info, move_info):
  """Предложение ответить на вопрос по коду"""
  return BK_calc.calc_bk(
    game_info['player_memory']['code'], move_info)
```

И, наконец, самый главный модуль — `BK`:

Листинг 4.13. Файл BK.py. Модуль `BK`

```
def game(game_name, game_format,
         player1_name, player2_name,
         player1_module, player2_module):
  """game(game_name,  -- название игры
         game_format, -- кортеж. 0-й элемент -- размер шифра
         player1_name, player2_name, -- имена игроков
```

```
        player1_module, player2_module) -- модули игроков
    -> результат_поединка

Координация игры двух игроков.
"""
# инициализируем структуру информацией об игре
p1 = { 'game_name': game_name,
       'game_format': game_format,
       'game_history': [],
       'name': player1_name, }
p2 = { 'game_name': game_name,
       'game_format': game_format,
       'game_history': [],
       'name': player2_name, }
# инициализируем игры
player1_module.start_game(p1)
player2_module.start_game(p2)
# еще никто не выиграл:
p1_won = 0
p2_won = 0
# основной цикл игры, пока один из игроков не победит
while not (p1_won or p2_won):
  # просим игроков делать ходы
  p1_move = player1_module.make_move(p1)
  p2_move = player2_module.make_move(p2)
  # просим игроков ответить
  p1_answer = player1_module.give_answer(p1, p2_move)
  p2_answer = player2_module.give_answer(p2, p1_move)
  # ходы вошли в историю игры
  p1['game_history'].append((p1_move, p2_answer))
  p2['game_history'].append((p2_move, p1_answer))
  print p1_move, p2_answer, ":", p2_move, p1_answer
  # проверяем, кто угадал шифр
  p1_won = (p2_answer == (4, 0))
  p2_won = (p1_answer == (4, 0))

if p1_won and p2_won: return "ничья"
elif p1_won: return "победил "+ player1_name
```

```
  elif p2_won: return "победил "+ player2_name
  else: return "игра прервана"

def test():
  """test()

  Тест модуля BK.py
  """
  import BK_human_player, BK_computer_player

  print game("Тестовая игра", (4,), "Комп1", "Комп2",
             BK_computer_player, BK_computer_player)
  # print game("Тестовая игра", (4,), "Игрок1", "Компьютер",
  #            BK_human_player, BK_computer_player)
  # print game("Тестовая игра", (4,), "Игрок1", "Игрок2",
  #            BK_human_player, BK_human_player)

if __name__ == "__main__":
  test()
```

Запустив этот модуль (разумеется, файлы с другими модулями должны быть в том же каталоге), мы станем свидетелями поединка компьютера с самим собой:

```
[9, 7, 8, 1] (0, 1) : [8, 6, 5, 3] (0, 2)
[4, 2, 5, 9] (0, 2) : [5, 7, 8, 4] (1, 1)
[5, 3, 2, 7] (0, 2) : [3, 8, 1, 4] (0, 1)
[1, 4, 3, 5] (0, 2) : [0, 5, 8, 2] (0, 2)
[0, 5, 1, 2] (3, 0) : [5, 0, 3, 7] (0, 1)
[6, 5, 1, 2] (4, 0) : [2, 7, 6, 8] (2, 2)
победил Комп1
```

Начало игры с человеком может выглядеть так:

```
Игрок1!
Задумайте шифр для игры 'Тестовая игра'.
Тестовая игра
Игрок1, Ваш ход: 1234
Игрок1 !
Сколько быков, коров в 3850? 2,0
[1, 2, 3, 4] (0, 1) : [3, 8, 5, 0] (2, 0)
```

```
Тестовая игра
1234  --  0 1
Игрок1, Ваш ход:
```

Упражнение

Попробуйте разработать модули для игры "Морской бой" по составленным ранее описаниям интерфейсов.

4.5. Итоги нашего проекта

Проект программы для игры "Быки и коровы" готов и даже работает, поэтому самое время подвести некоторые итоги.

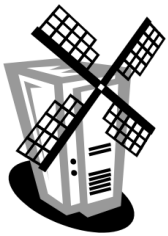
Само разбиение проекта на модули могло с первого раза состояться не так, как хотелось бы. У автора этой книги первый вариант разбиения на модули получился не очень удачным. К сожалению, это стало понятно уже тогда, когда все модули были написаны. Промах заметен в модуле `BK_computer_player`, который фактически является неким переходником между более удачным вариантом интерфейса между `BK` и нижележащими модулями и старыми интерфейсами модулей `BK_init`, `BK_calc` и `BK_strategy`. Можно сказать, что данная ситуация является очень типичной в реальном программировании: новое может прийти на смену старому постепенно, используя старые интерфейсы и хорошо отлаженные модули.

Упражнение

Проанализируйте дизайн всего проекта еще раз. Как бы вы теперь разбили проект на модули, если бы его нужно было делать с нуля?

Автору было сложно написать программу "Быки и коровы" без использования объектно-ориентированного программирования (которое мы изучим только в следующей главе). Обязательно прочитайте следующую главу и вернитесь к этой, чтобы увидеть, что именно можно упростить в проекте "Быки и коровы".

Глава 5



Объектный подход

Реализация достаточно сложного проекта обычно включает в себя *анализ* предметной области, *дизайн (проектирование)* и собственно программирование. Рассмотренный в предыдущей главе модульный подход является основой для еще более интересного и очень популярного *объектно-ориентированного анализа (ООА)*, *дизайна (ООД)* и *программирования (ООП)*.

Стадии анализа и дизайна, разумеется, не зависят от языка программирования, тогда как объектно-ориентированное программирование удобно вести с помощью объектно-ориентированного языка программирования.

Python с самого начала задумывался как объектно-ориентированный язык, и все встречающиеся в нем величины от чисел до модулей являются *объектами*. А что такое объект с точки зрения объектно-ориентированного подхода, мы узнаем из следующего раздела.

Замечание

При рассмотрении объектно-ориентированного подхода нередко используются различные виды схем, диаграмм, иллюстраций. Разработаны даже целые языки диаграмм, такие как UML (Universal Modelling Language, универсальный язык моделирования). Однако мы решили не использовать никаких графических иллюстраций объектного подхода в этой главе, т. к. считаем, что язык Python достаточно нагляден сам по себе. Схемы и диаграммы зачастую дают лишь иллюзию понимания и, по нашему мнению, не годятся для начального изучения объектно-ориентированного подхода, притупляя столь необходимое программистам воображение. Это никоим образом не означает, что схемы и диаграммы вообще бесполезны. Более того, если вам удобно обозначать зависимости между классами и объектами графически, обязательно делайте это, разбирая приведенные примеры, выполняя упражнения или решая свои задачи.

5.1. Терминология

Каждый, наверное, интуитивно представляет себе, что такое объекты в реальной жизни. Практически все, на что ни упадет взгляд, является физическим объектом: от маленького камешка до НЛО. В этом разделе мы дадим определения большинства ключевых понятий объектного подхода.

Различные авторы по-разному определяют понятие объекта, однако все они в той или иной степени сходятся на том, что программные объекты анало-

гичны физическим и часто являются их моделями. Именно поэтому объектно-ориентированный подход является во многих случаях естественным: человек привык отличать объекты в реальной действительности, и программист — системный аналитик применяет это умение при анализе предметной области для выделения в ней объектов. Эти объекты, скорее всего, будут использованы при объектно-ориентированном дизайне и найдут свое отражение при объектно-ориентированном программировании в виде программных объектов. В этой книге мы называем программные объекты просто объектами, т. к. речь почти все время идет о них.

Когда мы выделяем объекты, мы обычно абстрагируемся (отвлекаемся) от большинства их свойств, концентрируясь на существенных для нашей задачи свойствах. *Абстрагирование* — один из ключевых принципов объектно-ориентированного подхода. В результате абстрагирования мы получаем абстракцию — то, что мы называем объектом. Но теперь нам проще с ним совладать: ведь из всего многообразия свойств и образцов поведения мы выбрали только те, которые касаются решаемой задачи.

С точки зрения объектно-ориентированного подхода объект — нечто, обладающее *значением (состоянием)*, *типом (поведением)* и *индивидуальностью.*

Легче всего пояснить понятие объекта с помощью аналогии с физическим объектом. Так, с точки зрения электрической сети объект `Выключатель` имеет определенное состояние (включен или выключен). Конечно, `Выключатель` может иметь и другие характеристики: максимально допустимую силу тока и напряжение, цвет корпуса, вид использованного в контактах металла и т. п., что в нашей задаче несущественно. Под "поведением" `Выключателя` можно считать его возможность переходить из одного состояния в другое, т. е. `Выключатель` можно `включить` или `выключить`. И здесь мы отвлекаемся, абстрагируемся от некоторых видов поведения выключателя, например, от того, как действует выключатель механически: рубильник ли это, или выключатель с сенсорным управлением. Ну а индивидуальность нашего `Выключателя` позволяет нам иметь совершенно одинаковые `Выключатели` в разных комнатах, на кухне и в коридоре.

Объектами может быть не только то, что походит на реальные физические объекты. Отношения между объектами, процессы их взаимодействия, и даже более абстрактные "вещи" могут быть выделены в отдельные объекты.

Упражнение

Приведите несколько простых примеров объектов, участвующих в решении некоторой задачи. Укажите состояние и поведение объектов. От каких свойств и особенностей поведения объектов вы абстрагировались?

Конечно, в программе мы не станем описывать каждый `Выключатель` по-отдельности: мы опишем сразу класс выключателей.

Класс — множество объектов, имеющих общую структуру и поведение. С точки зрения языка Python класс — объект, служащий шаблоном для порождения своих экземпляров. Экземпляры класса, разумеется, тоже объекты. Благодаря тому, что они порождаются объектом-классом, они при своем рождении имеют общую структуру и поведение. Объект-класс записывается в программе на Python с помощью ключевого слова `class`. Для указанного выше примера с выключателем описание модели может выглядеть следующим образом:

```
class Switch:  # класс Выключатель
  def __init__(self, initial_state):  # инициализация выключателя
    self._state = initial_state       # в заданное положение
  def on(self):                       # включить
    self._state = 1
  def off(self):                      # выключить
    self._state = 0
  def toggle(self):                   # сменить состояние
    self._state = not self._state


s_kitchen = Switch(0)   # экземпляр Выключателя для кухни
s_room = Switch(0)      # комнаты
s_coridor = Switch(0)   # коридора
```

Здесь мы помимо имени класса (`Switch`) и его объектов (`s_kitchen`, `s_room`, `s_coridor`) видим и другие имена: `__init__`, `_state`, `on`, `off`, `toggle`. Указываемые этими именами объекты называются *атрибутами класса или объекта*. Определенные внутри оператора `class` функции называются *методами*. Теперь можно грамотно сказать, что состояние объекта определяет атрибут `_state`, а порождение и поведение объектов определяют методы `__init__()`, `on()`, `off()` и `toggle()`. Кстати, метод `init()` стоит особняком, т. к. отвечает за начальные установки объекта и его обычно называют *конструктором*. Последние три строчки предыдущего примера как раз и показывают вызовы конструкторов.

Рассмотрим теперь методы класса `Switch` поближе.

Метод `__init__()` принимает два аргумента. Первый аргумент любого метода принято называть в Python как `self` (сам), т. к. он всегда соответствует тому объекту, к которому применяется. Помимо этого метод `__init__()` принимает параметр `initial_state` для задания начального положения выключателя. Метод `__init__()` делает очень нехитрое дело: он присваивает атрибуту `_state` значение `initial_state`. Дело сделано — выключатель получает начальное значение.

Другие методы объектов класса `Switch` реализуют другие операции над объектом, или, как мы говорили выше, определяют его поведение. Методы можно также считать *сообщениями*, которые можно посылать объектам данного класса. Например, `Выключателю` можно сказать `Включись!` с помощью метода `on()`.

Ученые, развивающие объектно-ориентированное программирование, предлагают считать объекты черными ящиками, на корпусе которых есть только кнопочки для управления, да еще проводочки, идущие к другим объектам, которым данный объект может слать сообщения. Это еще один ключевой момент объектного подхода — *инкапсуляция*. При инкапсуляции, те, кто пользуются объектами, не должны нарушать их целостности, не должны видеть внутреннего устройства. В языке Python ящики не такие черные: мы отчетливо видим, что объект `Switch` имеет атрибут `_state`. Однако мы должны притвориться, что мы ничего такого не видим, и использовать для работы со `Switch` исключительно те атрибуты, которые не имеют одиночного подчеркивания "_" в начале своего имени:

```
#  продолжение примера
s_kitchen.on()   # выключатель_на_кухне, включись!
s_room.off()     # выключатель_в_комнате, выключись!
s_room.toggle()  # выключатель_в_комнате, смени положение!
```

Конечно, ничто не мешает нам разобрать наш `Выключатель` и поставить "жучка":

```
#  продолжение примера
s_coridor._state = 1
```

Но такое обращение нарушает "технику безопасности" при работе с объектами! Кстати, Python предоставляет некоторую защиту: попробуйте вместо имени `_state` использовать `__state`.

> ***Упражнение***
>
> Разработайте класс `Двойной_Выключатель` для моделирования выключателя, который обычно используют в трехрожковой люстре.

Приведенный нами пример напоминает скорее игру, чем серьезное программирование. При проектировании больших и сложных программных систем объекты перестают быть игрушками: в таких системах нам даже не охватить взором все объекты сразу, не говоря уже о том, чтобы для выполнения операций разбираться в их строении. В таких системах очень отчетливо проявляется еще один принцип построения объектно-ориентированных программ: объекты выстраиваются в *иерархию*.

В мире много различных примеров иерархий. От строения вещества: молекулы, атомы, элементарные частицы, и т. д. до чисто человеческих изобре-

тений: разбиение книги на части, главы, разделы, и т. д., деление страны на регионы, области, районы, и т. д. и т. п. Кстати, файлы в каталогах — тоже пример организации иерархии.

> ***Упражнение***
>
> Приведите другие примеры иерархий. Найдите примеры, в которых иерархии, на первый взгляд, не видны. Попробуйте рассмотреть такие примеры подробнее: может быть, не все объекты попали в поле вашего зрения.

В следующем разделе мы рассмотрим два варианта организации иерархии объектов: наследование и агрегацию.

5.2. Наследование и агрегация

В предыдущем разделе мы построили класс `Switch` (`Выключатель`), который позволял нам порождать объекты, управляющие состоянием конкретных выключателей. Теперь нам хотелось бы модернизировать некоторые выключатели, чтобы они могли показывать нам свое состояние, скажем, с помощью лампочки-индикатора. Во всем остальном они должны работать точно так же, как и раньше. Мы можем, конечно, используя возможности текстового редактора, скопировать определение класса, назвать его по-другому и добавить метод `indicator()`, показывающий состояние выключателя.

Но ООП предлагает другое решение, суть которого состоит в том, чтобы создать другой класс на базе существующего, наследовав все его методы. Приведем текст полностью:

```
class Switch:  # класс Выключатель
  def __init__(self, initial_state): self._state = initial_state
  def on(self): self._state = 1
  def off(self): self._state = 0
  def toggle(self): self._state = not self._state

class Switch_With_Indicator(Switch):  # Выключатель_с_индикатором
  def indicator(self): return self._state

s_coridor = Switch_With_Indicator(0)
print s_coridor.indicator()
s_coridor.toggle()
print s_coridor.indicator()
```

Результат выполнения:

```
0
1
```

Получается, что класс Switch_With_Indicator наследует свое поведение от класса Switch. Кроме того, мы добавляем метод indicator(). Таким образом, почти все операции над объектами класса Switch_With_Indicator определяются классом Switch. Класс Switch является *базовым классом* для класса Switch_With_Indicator. Объекты класса Switch_With_Indicator можно использовать везде, где применяются объекты класса Switch. Обратное не всегда верно, ведь если к объекту класса Switch применят метод indicator(), произойдет ошибка.

Упражнение

Создайте класс Button (Кнопка), который бы явился базовым для класса Switch, т. к. не предоставлял бы методов для включения on() и выключения off(), а только метод для переключения toggle().

А что если нам нужна Кнопка с индикатором? Все правильно:

```
class Button_With_Indicator(Button):
  def indicator(self): return self._state
```

Получается, что нам приходится дважды записывать в программе определение метода indicator(): отдельно для Button_With_Indicator и Switch_With_Indicator. Вот тут мы начинаем понимать, что Индикатор удобнее считать отдельным объектом и включать его при необходимости в Кнопки и Выключатели. То есть, например, Выключатель_с_индикатором является комбинацией собственно механизма включения/выключения и индикатора.

Сделать это можно двумя способами. Первый способ использует *множественное наследование*: мы создаем отдельный класс для Индикатора (Indicator), а класс Switch_With_Indicator получаем как результат наследования от классов Switch и Indicator:

```
class Indicator:
  """класс Индикатор"""
  def indicator(self): return self._state


class Switch:
  """класс Выключатель"""
  def __init__(self, initial_state): self._state = initial_state
  def on(self): self._state = 1
  def off(self): self._state = 0
  def toggle(self): self._state = not self._state

class Switch_With_Indicator(Switch, Indicator):
  """класс Выключатель с Индикатором"""
  pass
```

```
class Button_With_Indicator(Button, Indicator):
  """класс Кнопка с Индикатором"""
  pass
```

Получается, что `Switch_With_Indicator` является смесью `Switch` и `Indicator`. Класс `Indicator` не имеет самостоятельного значения. Он существует ради одного метода — `indicator()`. Знатоки ООП советуют не злоупотреблять множественным наследованием, тем более что в данном примере того же эффекта можно добиться и не создавая дополнительного класса. Второй способ таков:

```
def m_indicator(self):
  """метод для Индикатора"""
  return self._state

class Switch_With_Indicator(Switch):
  """класс Выключатель с Индикатором"""
  indicator = m_indicator

class Button_With_Indicator(Switch):
  """класс Кнопка с Индикатором"""
  indicator = m_indicator
```

Упражнение

Придумайте, какие еще "украшения" можно добавить к классу `Кнопка`? Придумав, напишите их реализацию на Python.

Другой подход для создания подобных комбинаций — *агрегация*. Говоря упрощенно, при агрегации объект содержит другие объекты, например, в качестве атрибутов. Разумеется, при этом в классе, в котором применяется агрегация, нужно предусмотреть методы для работы с содержащимися в нем объектами. В некоторых случаях хранение других объектов является основной задачей класса. Такой класс называют *контейнером* (от англ. слова *to contain* — хранить). Кстати, язык Python имеет несколько типов встроенных контейнеров — списки, кортежи, словари.

Следующий пример показывает, что объект класса `Sentence` (`Предложение`) может являться контейнером для объектов класса `Word` (`Слово`):

```
import re, string

class Word:
  def __init__(self, raw_word):
    self._word = raw_word
```

```
  def __str__(self):
    return self._word

class Sentence:
  def __init__(self, raw_sentence):
    raw_words = re.findall("[a-zA-Z]+", raw_sentence)
    print raw_words
    self._words = []
    for raw_word in raw_words:
      self._words.append(Word(raw_word))

  def __str__(self):
    return string.join(map(str, self._words))

print Sentence("This is example.")
```

5.3. Используем готовые объекты

Теперь мы уже знаем, что такое классы, объекты, атрибуты и методы. В этом разделе мы научимся использовать готовые объекты, предоставляемые различными стандартными модулями Python. Для работы с объектами того или иного класса нам необязательно смотреть на определение класса — достаточно знать его интерфейс и поведение объектов этого класса (эти вещи должна описывать документация). То есть нужно знать, какие параметры имеет конструктор класса, какие методы входят в интерфейс (и какие параметры они имеют), какие операции определены для данного класса, а также какие атрибуты разрешены для чтения или записи.

Работа с объектом начинается с его создания. Затем над объектом производятся операции (вызываются методы, читаются или записываются атрибуты). Другими словами, объект живет, принимает и отправляет сообщения другим объектам. Наконец, объект прекращает свое существование, когда он больше никому не нужен, и на него никто больше не ссылается. Некоторые объекты живут от начала работы программы до самого ее завершения, другие рождаются и умирают в рамках одной строчки программы. Кроме того, объекты могут жить и дольше самой программы, о чем мы поговорим в *разд. 5.6*.

Для создания экземпляра класса нужно вызвать класс и задать, если требуется, параметры. В качестве первого примера рассмотрим класс `StringIO`, определенный в одноименном модуле `StringIO`. Объекты этого класса ведут себя как обычные файловые объекты, работу с которыми мы изучали в *разд. 2.14*. Только ввод и вывод они осуществляют не в настоящие файлы, а

в строки. Модуль `StringIO` описан в *разд. 9.3.3*, здесь мы рассмотрим лишь некоторые методы.

Листинг 5.1. Работа с объектами класса `StringIO`

```
from StringIO import StringIO
f1 = StringIO()          # открываем "файл" для записи
# выводим в "файл" f1 числа от 0 до 7
for i in range(8):
  f1.write("%02i\n" % i)
# Теперь можно взять содержимое "файла" в виде строки:
content = f1.getvalue()
print content
# закрываем файл
f1.close()

f2 = StringIO(content)  # открываем "файл" для чтения из content
for line in f2.readlines():  # цикл по всем строкам файла f2
  print line,
f2.close()  # закрываем файл
```

Сначала мы создаем экземпляр класса `StringIO`, вызывая класс `StringIO` подобно функции. Далее мы пишем в f1, как будто это обычный файл, применяя метод `write()`. После этого, с помощью специального метода `getvalue()` (он отсутствует у обычных файловых объектов) строка `content` получает содержимое файла f1. Файл f1 закрывается с помощью метода `close()`.

Во второй части приведенного примера мы создаем файл f2 для чтения из строки `content`. Затем с помощью `readlines()` читается список строк файла f2, который тут же используется в цикле `for`. После того как выведены все строки, файл f2 закрывается.

> ***Упражнение***
>
> Составьте программу, которая прочитает строки заданного в программе текста и выведет их оператором `print`. Указание: для чтения строки из файла применяется метод `readline()`, *см. разд. 2.14*.

В следующем примере мы познакомимся с другими часто встречающимися на практике объектами — регулярными выражениями (шаблонами) и результатами сравнения. Мы уже не раз использовали регулярные выражения, обходя молчанием "странный" способ доступа к результатам сопоставления с шаблоном.

Для работы с регулярными выражениями мы можем использовать функции: `re.search()` (найти), `re.match()` (сопоставить), `re.split()` (разбить) и т. д. Однако для более эффективной работы, когда нам требуется применять одно и то же регулярное выражение много раз, можно подготовить (скомпилировать) регулярное выражение с помощью функции `re.compile`, создав специальный объект — `RegexObject`. Этот объект можно затем использовать для всех сопоставлений строк с данным шаблоном.

Результаты сопоставления представляются в виде другого объекта — `MatchObject`. Этот объект содержит всю необходимую информацию о результатах сопоставления.

Рассмотрим следующий пример:

```
from re import compile, MULTILINE
text = """План
1. Первый пункт
2. Второй пункт
3. Третий пункт
"""

# Порождаем Regex объект r:
r = compile("^(\d+)\.\s(.*)$", MULTILINE)

# Удачный поиск приводит к созданию Match-объекта:
m = r.search(text)
# Если поиск удачен:
if m:
  # С помощью методов объекта m можно получить различную информацию
  # о произведенном сравнении:
  print "Весь найденный фрагмент:", m.group()
  print "Первая группа (\d+):", m.group(1)
  print "Заданные группы:"
  for g in m.group(1, 2, 0):
    print "   ", g
  print "Группа 2 начинается с позиции:", m.start(2)
  print "Группа 2 заканчивается до позиции:", m.end(2)
  print "Теперь вместе:", m.span(2)
  print "Проверим:", text[m.start(2):m.end(2)]
```

Результат выполнения:

```
Весь найденный фрагмент: 1. Первый пункт
Первая группа (\d+): 1
```

```
Заданные группы:
    1
    Первый пункт
    1. Первый пункт
Группа 2 начинается с позиции: 8
Группа 2 заканчивается до позиции: 20
Теперь вместе: (8, 20)
Проверим: Первый пункт
```

Сначала с помощью функции `compile()` из модуля `re` мы получаем `Regex`-объект. Этот объект имеет несколько методов (`match()`, `search()`, `split()`, `sub()` и т. п.), но нас сейчас заинтересовал только `search()` — поиск. Результат сопоставления текста `text` с шаблоном, который "зашит" в объекте `r`, дается методом `search()` в виде `Match`-объекта. Если бы сравнение не удалось (в тексте не нашлось бы фрагмента, удовлетворяющего шаблону), метод `search()` выдал бы `None`. Чтобы исключить эту ситуацию, поставлена соответствующая проверка — оператор `if`. Теперь можно применять различные методы `Match`-объекта, чтобы узнать побольше о свершившемся сопоставлении. Из примера все это достаточно хорошо видно, если вспомнить, что группы выделяются скобками, а нулевая группа — это весь найденный фрагмент.

> ***Упражнение***
>
> Метод `search()` имеет необязательный второй параметр, который указывает, с какой позиции искать. Напишите программу, которая будет выводить все пункты плана, но без нумерации.

5.4. Строим свои классы

Мы уже рассмотрели несколько примеров и можем теперь попробовать составить собственную объектно-ориентированную программу.

В качестве задачи возьмем проблему тестирования знаний с помощью теста с вариантами ответов. Тестируемому (назовем его студентом) задаются вопросы из некоторого набора. На каждый вопрос предлагается несколько вариантов ответа. На основании анализа ответов на вопросы выставляется оценка.

Наверное каждый может представить себе процедуру экзамена, поэтому можно сразу после поверхностного анализа нашей предметной области выделить следующие объекты: `Тест`, `Студент`, `Набор_Вопросов`, `Вопрос`, `Формулировка_Вопроса`, `Вариант_Ответа`, `Правильный_Ответ`, `Ответ`, `Оценка` и `Экзамен()`.

Упражнение

Попробуйте провести подобный анализ для упрощенного аналога игры "Поле чудес", в которой один или несколько играющих отгадывают некоторое слово, по очереди предлагая по одной букве. Если буква встречается в загаданном слове, вместо точек, обозначавших неизвестные буквы, показываются отгаданные буквы, а отгадавший букву игрок получает случайное число очков от 1 до 10 включительно. Если буква не встречается в слове, ход переходит к следующему игроку. Игра заканчивается, когда кто-либо угадает последнюю букву в загаданном слове.

Теперь необходимо посмотреть, в каких отношениях находятся эти объекты и каким поведением они обладают. В табл. 5.1 представлен один из возможных вариантов взаимоотношений и поведения выделенных нами объектов.

Таблица 5.1. *Взаимоотношения и поведение объектов проекта "Экзамен"*

Класс объекта	Что представляет собой	Что умеет делать
`Тест`	Список объектов `Вопрос`	Дает любой вопрос из списка отобранных вопросов; выбирает заданное количество вопросов из заданного списка `Набор_Вопросов`
`Студент`	Интерфейс для общения с пользователем-студентом	Спрашивает имя; передает студенту сообщения; дает `Ответ` на `Вопрос`. Это единственный объект, который общается с пользователем программы
`Набор_Вопросов`	Список объектов `Вопрос`	Дает любой вопрос из списка
`Вопрос`	`Формулировка_Вопроса`, список объектов `Вариант_Ответа`, `Правильный_Ответ`	Представляет себя в виде текста; проверяет правильность заданного объекта `Ответ`
`Формулировка_Вопроса`	Строка с текстом вопроса	
`Вариант_Ответа`	Строка с текстом варианта ответа	
`Правильный_Ответ`	Строка с номером правильного ответа	
`Ответ`	Строка с номером ответа	
`Оценка`	Словарь расшифровки оценок	Рассчитывает оценку по заданному числу вопросов и правильных ответов; представляет оценку в текстовом виде
`Экзамен()`	`Тест`	Проверяет знания объекта `Студент`, возвращая объект `Оценка`

Думается, составление подобной таблицы не затруднило бы читателя, незнакомого с объектным подходом к проектированию: ведь мы, по существу, выделили и описали объекты, которые достаточно хорошо знакомы всем, кто сдавал экзамен в форме теста.

Упражнение

На основании ранее сделанного анализа для игры "Поле чудес" составьте таблицу взаимоотношений и поведения объектов.

Как видно из приведенной таблицы, для половины наших объектов мы обошлись встроенными объектами языка Python — строками и списками. Сделаем еще один шаг и напишем интерфейсы для наших классов (имена с подчеркиваниями будут объяснены после листинга):

Листинг 5.2. Классы проекта "Экзамен"

```
class Student:
  """Студент"""
  def __init__(self): """знакомимся со студентом"""
  def listen(self, message): """доводим сообщения до студента"""
  def answer_question(self, question):
    """получить ответ от студента на вопрос question"""


class Exam:
  """Экзамен"""
  def __init__(self, test): """экзамен начинается"""
  def test_skills(self, student): """проверка знаний студента"""

class Test:
  """Тест"""
  def __init__(self, questions, number=0): """готовим тест"""
  def __len__(self): """количество вопросов в тесте"""
  def __getitem__(self, n): """взятие n-го вопроса"""
  def Pick(self, questions): """выбираем вопросы"""

class Question:
  """Вопрос"""
  def __init__(self, text, answers, right_answer):
    """инициализируем вопрос"""
  def __str__(self): """представляем вопрос в виде текста"""
```

```
  def check_answer(self, given_answer):
    """проверить правильность ответа"""


class Mark:
  """Оценка"""
  def __init__(self, answered, total): """инициализируем оценку"""
  def __str__(self): """текстовая оценка"""
```

Помимо конструкторов классов, которые, как мы уже знаем, записываются как `__init__`, в приведенном коде на Python используются и другие имена: `__str__`, `__len__`, `__getitem__`. Методы, обозначенные такими именами, имеют специальный смысл и позволяют объектам определять операции, подобные операциям над встроенными объектами. Например, класс `Test` определяет метод `__len__()`, и потому можно узнавать количество вопросов в тесте с помощью встроенной функции `len()`. Метод `__str__()` используется для преобразования объекта к строке. К примеру, класс `Question` определяет метод `__str__()`, в котором формируется строка с текстом вопроса. Таким образом, объект класса `Question` можно легко вывести с помощью оператора `print` или применить функцию `str()` для преобразования этого объекта к строке. Метод `__getitem__()` позволяет обращаться к объекту с помощью обычной для последовательностей операции взятия объекта по индексу: `test[2]` будет означать, что мы хотим третий вопрос (нумерация вопросов начинается с нуля) данного теста. Кроме того, наличие метода `__getitem__()` позволяет объекту использоваться как последовательность в операторе `for`, например:

```
for question in test:
  print question
```

Этот пример, как вы наверное уже догадались, напечатает все вопросы теста. О том, как работает `__getitem__()`, можно справиться в *разд. 8.4.4.*

Упражнение

Запишите интерфейс классов для игры "Поле чудес".

Наш проект быстро и незаметно приблизился к стадии собственно кодирования. Следующий листинг представляет законченный вариант проекта "Экзамен":

Листинг 5.3. Файл exam.py. Реализация проекта "Экзамен"

```
#!/usr/bin/python
"""Проект "Экзамен" """
from random import choice
```

```
class Student:
  """Студент"""
  def __init__(self):
    """знакомимся со студентом"""
    self._name = raw_input("Как вас зовут? ")
  def listen(self, message):
    """доводим сообщения до студента"""
    print self._name+",", message
  def answer_question(self, question):
    """получить ответ от студента на вопрос question"""
    print question
    answer = raw_input("Ваш вариант ответа: ")
    return answer

class Exam:
  """Экзамен"""
  def __init__(self, test):
    """экзамен начинается"""
    self._test = test
  def test_skills(self, student):
    """проверка знаний студента"""
    student.listen("ответьте на вопросы")
    sum = 0
    total = 0
    for question in self._test:
      total = total+1
      if question.check_answer(student.answer_question(question)):
        sum = sum+1
    return Mark(sum, total)

class Test:
  """Тест"""
  def __init__(self, questions, number=0):
    """готовим тест"""
    # кол-во вопросов в тесте задано либо берем все
    self._number = number or len(questions)
    self.Pick(questions)
  def __len__(self):
    """количество вопросов в тесте"""
```

```
    return len(self._questions)
  def __getitem__(self, n):
    """взятие n-го вопроса"""
    return self._questions[n]
  def Pick(self, questions):
    """выбираем вопросы"""
    self._questions = []
    for i in xrange(self._number):
      picked = choice(questions)
      self._questions.append(picked)
      questions.remove(picked)

class Question:
  """Вопрос"""
  def __init__(self, text, answers, right_answer):
    """инициализируем вопрос"""
    self._text = text
    self._answers = answers
    self._right_answer = right_answer
  def __str__(self):
    """представляем вопрос в виде текста"""
    text = str(self._text)
    num = 0
    for a in self._answers:
      num = num+1
      text = """%(text)s\n%(num)s. %(a)s""" % vars()
    return text
  def check_answer(self, given_answer):
    """проверить правильность ответа"""
    return self._right_answer == given_answer

class Mark:
  """Оценка"""
  mark_names = {2: 'неудовлетворительно', 3: 'удовлетворительно',
      4: 'хорошо', 5: 'отлично'}
  def __init__(self, answered, total):
    """инициализируем оценку"""
    self._mark = max(2, answered*5/total)  # меньше двойки не ставим
  def __str__(self):
```

```
    """текстовая оценка"""
    return str(self.mark_names[self._mark])

def module_test():
  test = Test([
    Question("2*2=", ["2", "22", "4", "5"], "3"),
    Question("7*7=", ["77", "49", "14", "4"], "2"),
    Question("3*5=", ["20", "30", "15", "12"], "3"),
    ], 2)

  student = Student()
  exam = Exam(test)
  mark = exam.test_skills(student)
  print mark
if __name__ == "__main__":
  module_test()
```

Заметьте, что мы реализовали проект "Экзамен" в виде модуля `exam`. Пример его использования приведен в функции `module_test()`. Набор вопросов состоит из трех вопросов, а студенту задается только два вопроса.

> ***Упражнение***
>
> Разберитесь в реализации модуля `exam` и попробуйте реализовать игру "Поле чудес" на основе всех выполненных ранее упражнений.

В этом разделе мы выполнили анализ (выделили классы и их взаимоотношения), проектирование (конкретизировали интерфейсы классов) и программирование (написали модуль `exam`). Обычно жизнь программного продукта на этом не заканчивается, и в следующем разделе мы рассмотрим, как можно модифицировать наш проект.

5.5. Модификация объектного проекта

В нашем быстро меняющемся мире практически невозможно выполнить ни один программный проект, что называется, "на века". Рано или поздно появляется необходимость модификации существующей программы. В этом разделе мы модифицируем наш проект "Экзамен" так, чтобы он позволял генерировать вопросы по заданному шаблону.

Для достижения поставленной цели нужно, прежде всего, определить процедуру генерации вопроса, а затем создать на основе класса `Question` (`Вопрос`) новый класс, который бы использовал эту процедуру. В общем случае

автоматическая генерация вопроса очень сложна, поэтому мы ограничимся темой "Таблица умножения".

Оказывается, что для обеспечения автоматической генерации вопроса нам нужно изменить только один метод класса `Question`, а именно, метод `__init__()`. На следующем листинге представлен модифицированный вариант проекта "Экзамен", который добавляет к старому проекту возможность автоматической генерации вопросов по теме "Таблица умножения":

Листинг 5.4. Модификация проекта "Экзамен"

```
from exam import *
import random
class Mul_Question(Question):
  """Вопрос по таблице умножения"""
  def __init__(self, text, answers, right_answer):
    """инициализируем вопрос
    text теперь шаблон вопроса, а
    answers теперь не список, а кол-во вариантов ответа
    """
    x = random.randint(1, 10)  # первый множитель
    y = random.randint(1, 10)  # второй множитель
    a = x*y             # произведение
    self._text = text % vars()  # наполняем шаблон значениями
    self._right_answer = right_answer
    variants = range(max(1, a-20), a) + range(a+1, min(a+20, 100))
    self._answers = []
    while len(self._answers) < answers and variants:
      if len(self._answers) == int(right_answer)-1:
        self._answers.append(a)
      else:
        c = random.choice(variants)
        self._answers.append(c)
        variants.remove(c)

test = Test([
  Mul_Question("%(x)s * %(y)s =", 4, "3"),
  Mul_Question("%(x)s * %(y)s =", 4, "2"),
  Mul_Question("%(x)s * %(y)s =", 4, "3"),
  ], 2)
```

```
student = Student()
exam = Exam(test)
mark = exam.test_skills(student)
print mark
```

Следует заметить, что интерфейс класса `Mul_Question` отличается от интерфейса `Question` тем, что вместо списка вариантов ответа передается количество вариантов ответа. Кроме того, аргумент `text` является строкой форматирования для текста вопроса, а не самим текстом вопроса.

> ***Упражнение***
>
> Модифицируйте свой проект "Поле чудес" таким образом, что игроку иногда выпадает удача открыть любую букву в загаданном слове. Сколько методов в скольких классах вам пришлось изменить для реализации этого нового правила игры?

Итак, один из вариантов расширения функциональности объектного проекта — наследование новых классов из уже созданных с целью замены некоторых методов или дополнения новых методов. Разумеется, объектный подход сам по себе не гарантирует от ошибок. Например, если бы мы убрали из класса `Mul_Question` метод `check_answer()`, то объекты класса `Mul_Question` уже не смогли бы взаимодействовать с объектами других классов так же успешно, как объекты класса `Question`. С другой стороны, мы могли бы совершенно переделать класс `Question`, сохранив его интерфейс, и объекты этого переделанного класса смогли бы работать с другими объектами так же, как и раньше. Таким образом, при использовании объектного подхода во многих случаях проект легче подвергать модификации, чем при использовании традиционного структурно-модульного подхода, который мы рассматривали в предыдущих главах.

5.6. Устойчивые объекты

При работе программы объекты, с которыми она имеет дело, хранятся в оперативной памяти. С завершением программы они, естественно, пропадают без следа. В некоторых случаях необходимо, чтобы объект жил дольше, чем использующая его программа, и мог быть передан по сети или сохранен в базе данных. Все эти возможности напрямую связаны с *устойчивостью объектов*, т. е. сохранением состояния объекта во времени и пространстве.

Для обеспечения устойчивости объектов в Python используется модуль `pickle`. Название происходит от англ. *to pickle* — солить, мариновать, в общем, консервировать. "Законсервировать" можно почти любой объект (тех, кого интересует применимость `pickle`, могут обратиться к *разд. 9.2.7*). Сле-

дующий простой пример показывает, как сохранить объект в файле (для работы этого примера требуется модуль `exam`, сделанный нами в *разд. 5.4*).

Строим тест для экзамена и сохраняем его в файле:

Листинг 5.5. Сохранение объекта `exam` в файле

```
import pickle
from exam import *

test = Test([
  Question("2*2=", ["2", "22", "4", "5"], "3"),
  Question("7*7=", ["77", "49", "14", "4"], "2"),
  Question("3*5=", ["20", "30", "15", "12"], "3"),
], 2)

exam = Exam(test)

file = open("test.dat", "wb")
pickle.dump(exam, file)
file.close()
```

Берем объект `exam` из файла и как ни в чем не бывало проводим экзамен:

Листинг 5.6. Загрузка экзамена из файла и запуск

```
import pickle
from exam import *

file = open("test.dat", "rb")
exam = pickle.load(file)
file.close()

student = Student()
mark = exam.test_skills(student)
print mark
```

Теперь можно легко себе представить, что первую программу запускает преподаватель на своем компьютере, а вторую — студент на своем, после получения объекта `exam`, например, по электронной почте.

Упражнение

В нашем примере в объекте `exam` присутствуют и ответы на вопросы, что, разумеется, не очень годится для проведения дистанционного экзамена. Предложите модификацию классов проекта "Экзамен", которая позволит провести экзамен не опасаясь, что студент подсмотрит ответы.

5.7. Что дальше?

В этой главе вы получили представление об объектно-ориентированной технологии создания программ. Для небольших задач объектно-ориентированный подход может оказаться не совсем уместным, однако без понимания таких понятий, как "класс", "объект", "атрибут", "метод" и умения использовать готовые объекты в своих программах вряд ли можно обойтись при использовании современных языков и систем программирования. Язык программирования Python позволяет без лишних деталей и тонкостей создавать свои классы. Если вы хотите получить действительно удобное описание вашей задачи, необходимо хорошо продумать взаимоотношения между объектами предметной области.

Глава 6



Графический интерфейс пользователя

Программирование *графического интерфейса пользователя (GUI, Graphical User Interface)* знакомит нас с еще одной разновидностью программ — *событийно-управляемые (event-driven) программы*. Дело в том, что в основном графический интерфейс находится в ожидании действий пользователя, и программа с таким интерфейсом должна уметь реагировать на те или иные *события (events)*, которые происходят в связи с этими действиями. Пользователь щелкнул мышью по кнопке — одно событие, нажал клавишу <Enter> — это другое событие, и т. д. Для каждого события, на которое наша программа должна реагировать, мы пишем *обработчик события (event handler)*. В остальном событийно-управляемые программы ничем не отличаются от обычных, в них можно использовать все то, что мы изучали в предыдущих главах.

Изучая многочисленные возможности, которые предоставляет рассматриваемый графический пакет, не следует забывать, что искусство правильного построения человеко-машинного интерфейса состоит не в применении всего разнообразия имеющихся средств, а в тщательном выборе минимума средств, подходящих для решения данной задачи.

Прежде чем приступить к программированию графических интерфейсов, необходимо уточнить терминологию:

- *окна* — области экрана, обычно прямоугольной формы, на которых программа с графическим интерфейсом располагает свои визуальные компоненты;
- *виджеты* — визуальные компоненты графического интерфейса, с помощью которых пользователь имеет возможность управлять программой или наблюдать результаты ее работы. Примеры виджетов: кнопки, меню, поля ввода, текстовые поля и другие подобные "штучки" (виджет в переводе с английского — штучка);
- *рамки* — виджеты, основное назначение которых: содержать в себе другие виджеты. Расположение виджетов в рамке обычно подчиняется некоторым правилам;
- *дерево виджетов* — в GUI-программе вложение компонентов друг в друга можно изобразить в виде дерева, в корне которого — окно верхнего

уровня. Таким образом, у окна или виджета могут быть *дочерние виджеты*, вложенные в него, а он для них будет являться *родительским виджетом*.

Для наших построений мы будем использовать стандартный для Python модуль `Tkinter`. Это далеко не единственный выбор, но, пожалуй, самый логичный для начального изучения. Модуль `Tkinter` предлагает нам целый набор виджетов и достаточно простые средства для построения графических интерфейсов пользователя.

Замечание

Для понимания этой главы необходимо знать классы и объекты Python. О них разговор шел в предыдущей главе.

6.1. Простой пример

В качестве примера рассмотрим программу для вычисления площади круга. Программа должна иметь поле ввода радиуса, кнопки **Вычислить** и **Выход**, а также текстовое поле для размещения результата — вычисленной площади круга.

Листинг 6.1. Вычисление площади круга

```
from Tkinter import *
import math

def Square(radius):
  return math.pi * radius**2

def calculate_square():
  radius = float(radius_entry.get())
  try:
    square = "%11.3f" % Square(radius)
  except:
    square = "?"
  square_label.configure(text=square)

root = Tk()
root.title("Площадь круга")
frame = Frame(root)
radius_entry = Entry(frame, width=10)
```

```
radius_entry.grid(row=0, column=0)
square_label = Label(frame, text="?")
square_label.grid(row=0, column=1)
eval_button = Button(frame, text="Вычислить", width=10,
                     command=calculate_square)
eval_button.grid(row=1, column=0)
exit_button = Button(frame, text="Выход", width=10,
                     command=root.destroy)
exit_button.grid(row=1, column=1)
root.mainloop()
```

Сначала программа путем вызова конструктора класса `Tk` создает окно `root`. Окну присваивается имя **Площадь круга**. В окне `root` создается рамка `frame`. Далее определяются четыре используемых программой виджета: поле ввода радиуса `radius_entry`, текстовое поле для вывода площади круга `square_label`, кнопка `eval_button`, с которой связана команда вычисления площади `calculate_square()`, и кнопка `exit_button` для выхода из программы. Обратите внимание, что каждый виджет располагается в своей клетке, которая задается номером строки (`row`) и столбца (`column`) при использовании метода `grid()`. Кроме того, у всех виджетов задана их ширина `width`. После всех этих определений следует вызов метода `mainloop()` окна `root` для переключения программы в режим ожидания событий.

Событиями, обработка которых определена в этой программе, являются нажатия кнопок **Вычислить** и **Выход**. Для первого события вызывается функция `calculate_square()`, в которой вычисляется площадь круга, а для второго — метод `root.destroy()` окна `root` для закрытия этого окна (и завершения программы).

В функции `calculate_square()` используется метод `configure()` для изменения атрибута `text` объекта `square_label`. Разумеется, радиус сначала приходится превращать в число с плавающей точкой, а площадь круга представлять в виде строки с помощью операции форматирования.

На рис. 6.1 и 6.2 изображены окна программы "Площадь круга" сразу после запуска и после вычисления площади круга.

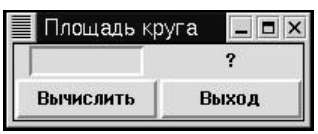


Рис. 6.1. Окно программы "Площадь круга" после запуска

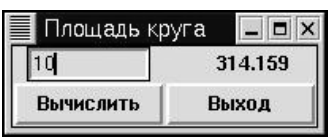


Рис. 6.2. Окно программы "Площадь круга" после вычисления площади

Наверное, было бы еще более естественно, если бы вычисления площади происходили после нажатия клавиши <Enter> (или <Return>) в поле ввода радиуса. Для этого придется связать событие "нажатие клавиши <Enter> с функцией `calculate_square`, т. е. добавить следующую команду после определения `radius_entry`:

```
radius_entry.bind("<Key-Return>", calculate_square)
```

Для достижения этого результата придется несколько переделать определение функции `calculate_square()`, чтобы она могла принимать необязательный параметр `event`, передаваемый подобным функциям при возникновении событий.

```
def calculate_square(event=None):
  radius = float(radius_entry.get())
  try:
    square = "%11.3f" % Square(radius)
  except:
    square = "?"
  square_label.configure(text=square)
```

После этих изменений программа "Площадь круга" будет вычислять площадь не только после нажатия кнопки **Вычислить**, но и при нажатии клавиши <Enter>.

> ***Упражнение***
>
> Постройте программу для вычисления объема цилиндра по аналогии с рассмотренной в этом разделе программой "Площадь круга".

6.2. Программа "Экзаменатор"

В этом разделе мы рассмотрим программу для проведения простейшего экзамена. Программа задает обучаемому несколько вопросов. Для каждого вопроса обучаемый должен выбрать один из вариантов ответа с помощью соответствующего переключателя и нажать кнопку **Ответ готов**. После того как программа получила ответы на все вопросы, она выдает количество правильных ответов.

В программе используется три вида кадров: заставка (рис. 6.3), кадр с вопросом (рис. 6.4) и кадр с результатом (рис. 6.5).

В листинге 6.2 приведен текст программы "Экзаменатор". Все данные, касающиеся конкретного экзамена, хранятся в файле examenator.dat. Для загрузки данных этот файл просто исполняется с помощью встроенной функции `execfile()`. В листинге 6.3 приведен пример экзамена из двух вопросов.

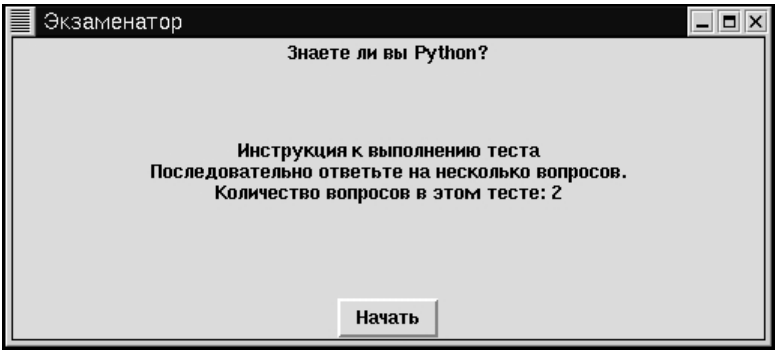


Рис. 6.3. Заставка (программа "Экзаменатор")

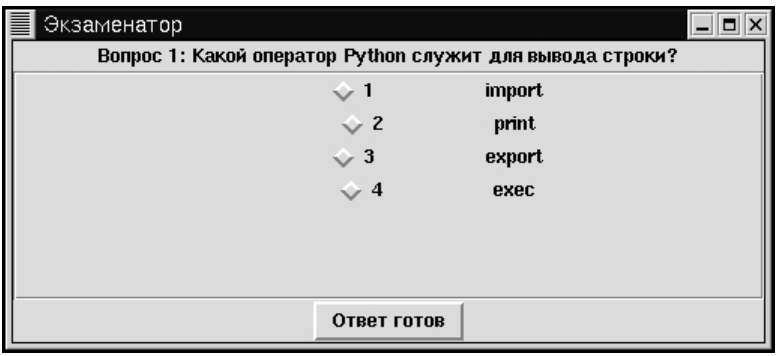


Рис. 6.4. Кадр с вопросом (программа "Экзаменатор")

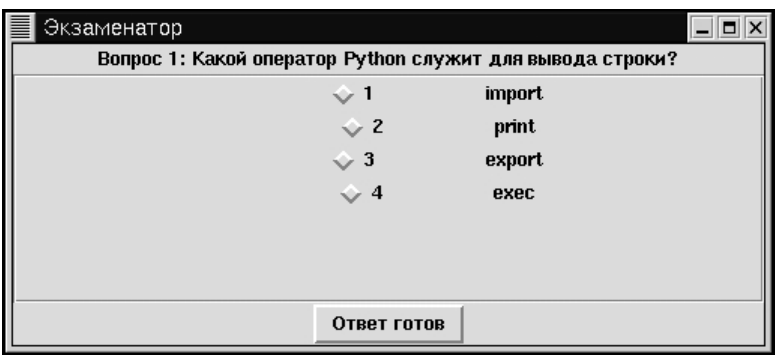


Рис. 6.5. Кадр с результатом (программа "Экзаменатор")

Далее порождается главное окно `root`, которое получает наименование "`Экзаменатор`" и минимальные размеры 500×200 пикселов. После этого определяются функции, которые служат для выполнения команд, связанных с кнопками на различных кадрах в программе.

Первый кадр программы получает имя `frame1`. Рассмотрим его подробнее:

```
frame1 = Frame(root, width=500, height=200)
```

Объект-рамка `frame1` получается в результате вызова конструктора класса `Frame`. Первым аргументом при вызове конструктора задается виджет-хозяин для вновь создаваемого виджета. В нашем случае это — главное окно программы. Кроме него можно задать несколько именованных аргументов, например ширину `width` и высоту `height`.

Примечание

К сожалению, `Tkinter` часто не считается с заданными размерами, пересчитывая их исходя из реальных потребностей вложенных виджетов. Именно поэтому следом идет вызов метода `propagate()` с нулевым аргументом, что отменяет нежелательный в нашем случае эффект.

Далее все происходит по уже знакомой схеме:

- новый виджет создается вызовом конструктора в виджете-хозяине. При этом можно задавать некоторые опции, используя именованные аргументы;
- вызывается метод менеджера расположения;
- объект-виджет может использоваться для создания других виджетов внутри себя или для индикации данных.

В нашем примере используется менеджер расположения *pack*. Ему тоже можно передавать некоторые опции, например `fill` (заполнение) для задания направлений, в которых данный виджет может использовать имеющееся место, и `expand`, который требует от менеджера расположения `pack` по возможности расширить виджет. Опция `side` позволяет указать сторону, к которой будет примыкать располагаемый виджет.

Опции можно задать и с помощью метода `config()`, как это делается, например, в функции `go_frame3()` для задания текста результата экзамена в виджете класса `Label` (этикетка).

В программе "Экзаменатор" используются следующие классы объектов:

- `Button` — кнопка;
- `Frame` — рамка;
- `IntVar` — переменная с целым значением;
- `Label` — этикетка (в нашей программе используется для вывода текста);
- `Radiobutton` — переключатель;
- `Tk` — главное окно Tk-программы;

Заметьте, что интерфейс программы становится доступным пользователю только после выполнения метода `mainloop()` — это в самом конце текста

программы. До этого момента все виджеты только инициализируются. Рамки `frame1`–`frame3` помещаются в главное окно не все сразу, а последовательно. При переходе на другой кадр с помощью функций `go_frame2()` и `go_frame3()` старая рамка "забывается" менеджером расположения (при выполнении метода `pack_forget()`), а на ее место устанавливается новая (с помощью метода `pack()`).

Первый кадр (в рамке `frame1`) имеет достаточно простую структуру: две этикетки (`title` и `instruction`) и кнопка **Начать** (`start`). Кнопка `start` имеет опцию `command`, в которой задана функция `go_frame2()`. Именно эта функция будет исполнена при нажатии на кнопку **Начать**.

Примечание

Объект-кнопка, фигурирующая в программе под именем `start`, отображается на экране в виде кнопки с надписью **Начать**.

Кадры с вопросами — самые сложные. Для этого вида кадра определена всего одна рамка `frame2`, но по ходу работы программы в присутствующие в ней этикетки записываются тексты для конкретных вопросов. В этикетке `ques_text` записывается текст вопроса, а кнопка `select`, видимая пользователю под именем **Ответ готов**, позволяет перейти к следующему вопросу (или к кадру с результатом, если вопрос был последним). Рамка `q_frame` содержит рамки с парами переключатель — текст варианта ответа. Словарь `answer_frame` как раз и дает объект-рамку по его номеру. Словарь `sel` — соответствующий переключатель, а `case` — этикетку с вариантом ответа. Эти словари в дальнейшем используются для изменения хранимых в виджетах значений. Кроме того, состояние группы переключателей взаимосвязано с переменной `sel_var`. Эта переменная содержит номер выбранного переключателями варианта ответа. Присваивание значения этой переменной приводит к изменению состояния переключателей, и, наоборот, изменение их состояния отражается в значении этой переменной. Получить значение переменной можно с помощью метода `get()` (см. определение функции `check_answer()`), а изменить — методом `set()` (см. `next_card()`).

При создании переключателей было задано несколько опций:

- `variable` — задает переменную, с которой связано состояние кнопки;
- `text` — задает надпись около кнопки;
- `value` — задает значение, соответствующее включенному состоянию кнопки. Именно его приобретает переменная в случае выбора кнопки пользователем.

Упражнение

Разберитесь, из каких виджетов состоит третий кадр программы "Экзаменатор".

Самая непростая часть программы — логика переходов между кадрами и связанные с этим действия. Функция `go_frame2()` осуществляет переход с первого кадра на кадр с вопросами. Функция `next_card()` заносит данные об очередном вопросе в предусмотренные для этого этикетки. В функции `check_answer()` проверяется правильность выбранного обучаемым варианта ответа (эта функция вызывается как из `next_card()`, так из `go_frame3()`). Заметьте, что переключатели сбрасываются присваиванием значения `-1` переменной `sel_var`: такое значение не соответствует ни одной кнопке и потому все связанные с этой переменной кнопки приходят в выключенное состояние.

Листинг 6.2. Файл examenator.py. Программа "Экзаменатор"

```
# загружаем все имена из модуля Tkinter
from Tkinter import *
# объявляем глобальные имена
global current_card, score

# материалы экзамена хранятся в виде программы на Python
# в файле examenator.dat
execfile("examenator.dat")

## главное окно ##
root = Tk()
root.title("Экзаменатор")
root.minsize(500, 200)

## Определяем функции для задания команд, связанных с кнопками
# Переход от заставки к первому вопросу
def go_frame2():
  frame1.pack_forget()
  frame2.pack(fill=BOTH, expand=1)
  next_card()

# проверка правильности данного ответа
def check_answer():
  global current_card, score
  student_answer = sel_var.get()
  right_answer = CARDS[current_card-1][2]
  if student_answer == -1:
```

```
    return
  if student_answer == right_answer:
    score = score + 1
  return 1

# Переход к следующему вопросу
def next_card():
  global current_card
  if current_card != 0:
    if check_answer() is None:
      return
  sel_var.set(-1)  # сбрасываем переключатели
  C = CARDS[current_card]
  i = 0
  for Q in C[1]:
    case[i].config(text=Q)
    i = i + 1
  current_card = current_card + 1
  if current_card == NCARDS:
    select.config(command=go_frame3)
  ques_text.config(text=C[0])

# переход от последнего вопроса к кадру с результатом
def go_frame3():
  global score
  if check_answer() is None:
    return
  answered = score
  total = NCARDS
  result.config(text=RESULT%vars())
  frame2.pack_forget()
  frame3.pack(fill=BOTH, expand=1)

## кадр 1 (заставка) ##
frame1 = Frame(root, width=500, height=200)
frame1.propagate(0)

title = Label(frame1, text=TITLE)
title.pack(fill=BOTH)
```

```
instruction = Label(frame1, text=INSTRUCTION%vars(), wraplength=480)
start = Button(frame1, text="Начать", command=go_frame2)
instruction.pack(fill=BOTH, expand=1)
start.pack()
frame1.pack(fill=BOTH, expand=1)

## кадр для задания вопроса ##
frame2 = Frame(root, width=500, height=200)
frame2.propagate(0)

ques_text = Label(frame2, text="", wraplength=480)
ques_text.pack(fill=BOTH)
answer_frame = {}
sel = {}
case = {}
sel_var = IntVar()        # переменная с выбранным ответом
q_frame = Frame(frame2, relief=GROOVE, border=2)
for i in range(ANSWERS):
    answer_frame[i] = Frame(q_frame)
    answer_frame[i].pack()
    sel[i] = Radiobutton(answer_frame[i], width=20,
                         variable=sel_var,
                         text=str(i+1),
                         value=i+1)
    sel[i].pack(side=LEFT)
    case[i] = Label(answer_frame[i], wraplength=460)
    case[i].pack(side=LEFT, fill=BOTH, expand=1)
q_frame.pack(fill=BOTH, expand=1)

select = Button(frame2, text="Ответ готов", command=next_card)
select.pack()

## кадр вывода результата ##
frame3 = Frame(root, width=500, height=200)
frame3.propagate(0)

result = Label(frame3, wraplength=480)
result.pack(fill=BOTH, expand=1)
```

```
quit = Button(frame3, text="Выход", command=root.destroy)
quit.pack()

current_card = 0
score = 0

root.mainloop()  # цикл обработки событий
```

Листинг 6.3. Файл examenator.dat. Данные для "Экзаменатора"

```
TITLE = "Знаете ли вы Python?"
ANSWERS = 4

CARDS = [
  ("Вопрос 1: Какой оператор Python служит для вывода строки?",
   ["import",
    "print",
    "export",
    "exec",
   ],
   2),
  ("""Вопрос 2: С помощью какой функции или операции
нельзя превратить число в строку?""",
   ["x % y",
    "str(x)",
    "max(x, y)",
    "repr(x)",
   ],
   3),
]

NCARDS = len(CARDS)
INSTRUCTION = """Инструкция к выполнению теста
Последовательно ответьте на несколько вопросов.
Количество вопросов в этом тесте: %(NCARDS)s
"""
RESULT = """Результат:
Вы ответили на %(answered)s вопросов из %(total)s."""
```

Упражнение

Если вы хорошо разобрались с темой предыдущей главы, посвященной объектному подходу к построению программ, попробуйте построить программу-экзаменатор с графическим интерфейсом на основе тех классов, которые были разработаны там же.

6.3. Игра "Сапер"

Наверное, многие знают логическую игру под названием "Сапер" ("Minesweeper"). Игроку предлагается разминировать поле `M` на `N` из клеточек, в некоторых из которых могут находиться мины. Игрок может либо пометить клеточку флажком (клетка заминирована), либо "наступить" на нее. В последнем случае случится одно из двух: если в клетке не было мин, в ней показывается общее количество мин в соседних клетках, иначе игра заканчивается поражением игрока. Для удобства при открытии клетки без единой мины в соседних клетках эти соседние клетки автоматически открываются. Нормальным завершением игры считается пометка флажками всех клеток поля, в которых находятся мины, и открытие всех остальных. Состояние игры индицирует лицо, которое напряжено, если игра не закончена, плачет, если игра проиграна, и смеется при выигрыше.

Для игры нам потребуются несколько пиктограмм, а именно: изображения цифр от одного до восьми, флажка, мины, а также три изображения лица с различными эмоциями: сосредоточенность, плач, улыбка. Пиктограммы должны быть одного размера, чтобы они могли помещаться в клетки поля.

В нашем упрощенном варианте игры в окне игры будут находиться следующие элементы:

- поле с клетками;
- счетчик числа флажков;
- лицо, мимикой выражающее ход игры.

Поле с клетками проще всего сделать, поместив в рамку (`Frame`) соответствующее количество кнопок. В кнопке можно располагать пиктограммы, а также управлять их "высотой".

В листинге 6.4 представлена программа для игры "Сапер". Вся логика игры заложена в классе `MineSweeper`. Этот класс наследован от класса `Frame` (рамка) и потому с ним можно обращаться как с виджетом (в частности, его можно поместить в окне верхнего уровня `root`).

Листинг 6.4. Файл ms.py. Программа игры "Сапер"

```
from Tkinter import *
import random
```

```
root = Tk()
root.title("Mine Sweeper")

class MineSweeper(Frame):
  """Класс с игрой Минер"""

  # подготовка объектов-изображений вида ?.ppm
  images = {}
  for i in "012345678BFNSCT":
    images[i] = PhotoImage(file=i+".ppm")

  # список приращений
  deltas = [(-1,-1), (-1,0), (-1,1),
            (0,-1), (0,1),
            (1,-1), (1,0), (1,1)]

  def __init__(self, master, N=10, M=10, MINES=12):
    """Инициализация объекта-игры"""
    # вызываем конструктор базового класса
    Frame.__init__(self, master)
    assert 0 <= MINES < N*M
    self.MINES = MINES  # количество мин
    self.N, self.M = N, M  # размер поля
    ff = Frame(self)  # рамка с лицом
    ff.pack()
    bf = Frame(self)  # рамка с полем кнопок
    bf.pack()
    self.face = Button(ff, image=self.images["T"])  # кнопка-лицо
    self.face.bind("<Button-1>", self.new_game)
    self.face.pack()
    self.button_field(bf)  # формируем поле кнопок
    self.new_game()  # новая игра

  def open(self, button):
    """Открывает клетку, соответствующую кнопке button"""
    x, y = self.bfield[button]
    if self.field[(x, y)][0] == 0:
      self.field[(x, y)][0] = 1
      if self.field[(x, y)][1]:
```

```
        self.game_state = 1
      elif self.field[(x, y)][2] == 0:
        for dx, dy in self.deltas:
          if self.bfield.has_key((x+dx, y+dy)):
            self.open(self.bfield[(x+dx, y+dy)])
      self.visited = self.visited + 1
    self.draw_cell(x, y)
    self.ready()
    self.show_state()

  def step(self, event):
    """Реакция на щелчок по кнопке"""
    self.open(event.widget)

  def flag(self, event):
    """Щелчок правой кнопкой - установка или снятие флажка"""
    x, y = self.bfield[event.widget]
    if self.field[(x, y)][0] == 0:  # если нет флажка
      self.field[(x, y)][0] = 2     # устанавливаем флажок
      self.visited = self.visited + 1
      self.flagged = self.flagged + 1
    elif self.field[(x, y)][0] == 2:  # иначе снимаем
      self.field[(x, y)][0] = 0
      self.visited = self.visited - 1
      self.flagged = self.flagged - 1
    self.draw_cell(x, y)
    self.ready()
    self.show_state()

  def ready(self):
    """Проверяет готовность поля и устанавливает соответствующее
    состояние"""
    if self.game_state != 1 \
       and self.visited == self.N*self.M \
       and self.flagged == self.MINES:
      self.game_state = 2

  def init_field(self):
    """Подготовка поля к игре"""
```

```
    # инициализация поля
    fld = {}
    for i in range(self.M):
      for j in range(self.N):
        fld[(i, j)] = [
          0,  # 0 - нетронутый участок, 1 - открытый, 2 - с флажком
          0,  # есть ли мина (0 или 1)
          0]  # количество мин вокруг (0-8)
    # установка мин
    nmines = 0
    while nmines < self.MINES:
      x = random.randrange(0, self.N)
      y = random.randrange(0, self.M)
      if fld[(x, y)][1] == 0:
        fld[(x, y)][1] = 1
        for dx, dy in self.deltas:
          if fld.has_key((x+dx, y+dy)):
            fld[(x+dx, y+dy)][2] = fld[(x+dx, y+dy)][2] + 1
        nmines = nmines + 1
    self.field = fld

  def draw_cell(self, X, Y):
    """Рисует клетку поля, изменяя параметры соответствующей кнопки"""
    b = self.bfield[(X, Y)]
    if self.field[(X, Y)][0] == 1:  # если участок открытый
      if self.field[(X, Y)][1] == 1:  # на нем мина
        img = self.images["B"]
      else:  # иначе пишем количество мин у соседей
        img = self.images[str(self.field[(X, Y)][2])]
      b.config(relief=FLAT, image=img)
    else:
      if self.field[(X, Y)][0] == 0:  # если участок нетронутый
        img = self.images["0"]
      else:  # если на участке флажок
        img = self.images["F"]
      b.config(relief=RAISED, image=img)

  def draw_field(self):
    """Рисует игровое поле"""
```

```
    for i in range(self.M):
      for j in range(self.N):
        self.draw_cell(i, j)
    self.show_state()

  def show_state(self):
    """Показывает мимику:
    T - сосредоточенность, C - плач, S - улыбка"""
    state = {0: "T", 1: "C", 2: "S"}[self.game_state]
    self.face.config(image=self.images[state])

  def button_field(self, frame):
    """Формирует поле из кнопок. Данные о поле хранятся в
    словаре bfield, из которого можно узнать по кнопке ее
    координаты и, наоборот, по координатам узнать кнопку."""
    self.bfield = {}
    img = self.images["0"]
    for i in range(self.M):
      for j in range(self.N):
        b = Button(frame, image=img, relief=RAISED,
                   highlightthickness=0, borderwidth=1)
        b.grid(row=i, column=j)
        b.bind("<Button-1>", self.step)  # щелчок левой кнопкой мыши
        b.bind("<Button-3>", self.flag)  # щелчок правой кнопкой мыши
        # записываем в словарь и координату
        self.bfield[(i, j)] = b  # кнопку по координатам
        self.bfield[b] = (i, j)  # координаты по кнопке

  def new_game(self, event=None):
    """Новая игра"""
    self.init_field()
    self.game_state = 0  # состояние игры:
    # 0 - в процессе, 1 - проигрыш, 2 - выигрыш
    self.flagged = 0  # количество флажков
    self.visited = 0  # количество тронутых участков
    self.draw_field()

game = MineSweeper(root)
game.pack()
root.mainloop()
```

Конструктор класса `MineSweeper` создает все необходимые рамки и кнопки. Заметьте, что в методе `__init__` вызывается и конструктор класса `Frame` (рамка). Метод `button_field()` формирует внутри заданной рамки кнопки, после чего метод `new_game` подготавливает поле к игре: инициализирует клетки игрового поля (они выполнены виджетами-кнопками) и расставляет мины. После этого программа выполняет цикл обработки событий `mainloop`, поэтому пользователь может нажимать на кнопки, вызывая соответствующие действия (рис. 6.6).

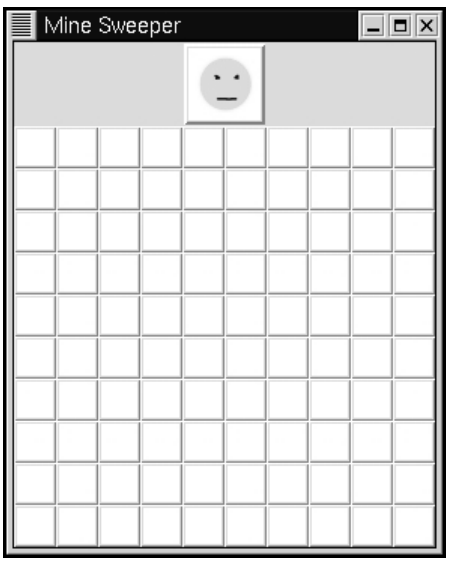


Рис. 6.6. Начальное состояние игры

При нажатии на кнопку с изображением лица игра сбрасывается в начальное состояние. Метод `step()` вызывается при нажатии пользователем левой кнопки мыши на одной из клеток поля. Из этого метода вызывается метод `open()`, который и выполняет некоторые действия, зависящие от состояния клетки игрового поля. Таких состояний может быть три: 0 — клетка не тронута, 1 — клетка открыта и 2 — на клетке поставлен флажок. Кстати, данные о клетках хранятся в словаре `field`, с помощью которого по координатам клетки (кортеж из двух целых чисел) можно получить кортеж из трех элементов. В первом элементе может быть `0`, `1` или `2` (как описано выше), во втором — `0` (мины нет) или `1` (мина есть), в третьем — суммарное количество мин в соседних клетках. Метод `open()` используется рекурсивно: если в данной клетке нет мины, он вызывается также и для всех соседей. Таким образом, объекты класса `MineSweeper` являются рамками, которые не только располагают внутри себя виджеты для игры "Сапер", но и реализуют

ее логику. На рис. 6.7 показаны ситуации проигрыша (было открыто поле с миной) и выигрыша (все клетки с минами отмечены флажками, а остальные открыты).

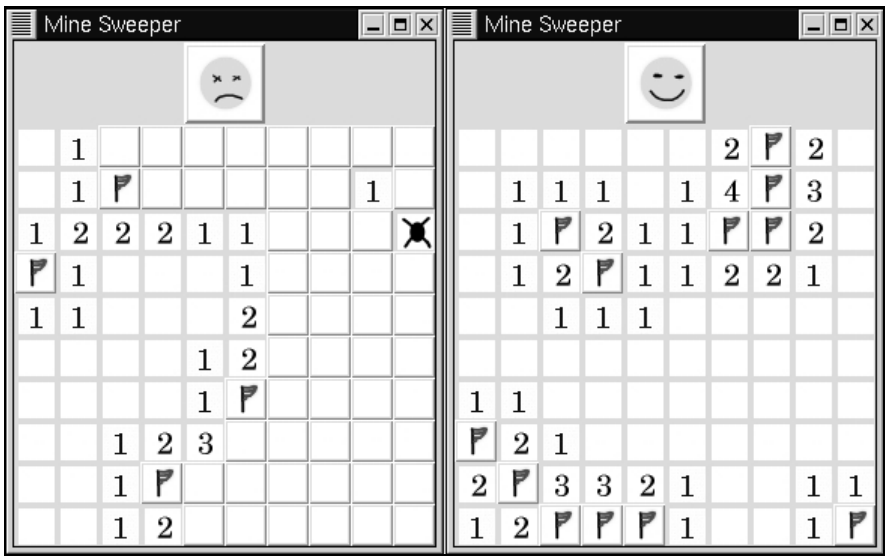
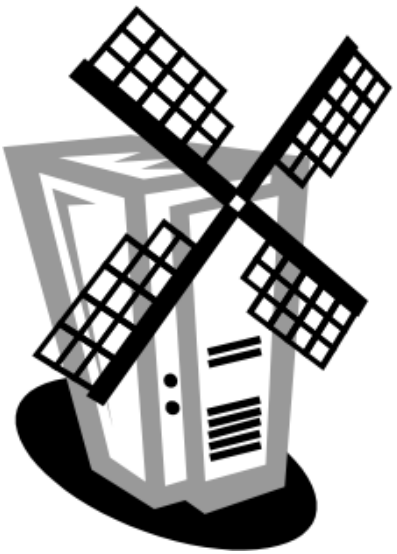


Рис. 6.7. Ситуации проигрыша и выигрыша

Упражнение

Добавьте в программу счетчик использованных флажков. В начале игры значение этого счетчика должно быть равно количеству мин на поле. В ходе игры значение счетчика должно уменьшаться до нуля. Попробуйте также добавить к программе главное меню.

Мы рассмотрели только три различных примера (не совсем доработанных) программ с графическим интерфейсом. Оставляем читателю возможность для упражнений. Однако хотелось бы еще раз напомнить одну из заповедей Python: "Простое лучше сложного" *(см. разд. 8.11)*. При создании графического интерфейса нужно думать прежде всего об удобстве, а не о количестве примененных "штучек".



Часть II

Описание языка Python

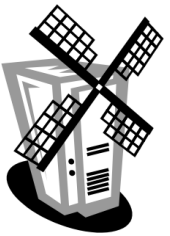
Эта часть — практически полное описание языка программирования Python и большинства стандартных модулей. Может использоваться как для углубленного изучения языка, так и в качестве справочника.

Глава 7. Общие сведения о Python

Глава 8. Описание языка

Глава 9. Основные стандартные модули

Глава 7



Общие сведения о Python

В этой главе даются некоторые полезные сведения о языке программирования Python. Кратко освещается его история, области применения, процедура установки и другие вопросы, связанные с этим программным продуктом.

7.1. История языка Python

Python ("Питон") — сравнительно новый язык программирования, разрабатываемый Гвидо Ван Россумом (Guido Van Rossum) с февраля 1991 года, когда он работал в CWI, известном как Stichting Mathematisch Centrum (см. **http://www.cwi.nl/**). Некоторые особенности Python позаимствовал из языка ABC, одновременно устранив некоторые присущие ABC проблемы. В то время Ван Россум работал в группе по распределенной ОС Amoeba и нуждался в языке, похожем на ABC, но с легким доступом к системным вызовам ОС Amoeba. В конце 1989 года Ван Россум решил спроектировать новый язык с большими возможностями к расширению, который он назвал Python. Имея опыт работы с Modula-2+, он применил в Python синтаксис и семантику исключений, заимствованные из Modula-3.

Python назван совсем не в честь змеи, а по названию комедийных серий 1970-х годов на телевидении BBC: "Monty Python's Flying Circus"[1]. По замыслу создателя, имя должно быть коротким, уникальным и чуть таинственным. Являясь почитателем теле-шоу, он посчитал, что Python — отличное имя для языка программирования.

В течение 1990-х годов Python развивался от начальной версии до версии 2.0, выпущенной в 2000 году. Самое активное (по числу новых строк в коде) развитие началось от версии 1.4. Переходы от версии к версии в основном не нарушали совместимости. Интересный курьез связан с версией 1.6, которая является развитием 1.5.2. Она вышла почти одновременно с действительно новой версией 2.0.

За время работы над Python его основной автор сменил несколько работодателей и сейчас вместе с другими разработчиками трудится в фирме Digital Creations.

---

[1] Летучий цирк Монти Питона — *Ред.*

7.2. О лицензиях

Python является одним из огромного числа успешных проектов, реализованных как *Open Source Software (программное обеспечение с открытым исходным кодом)* или *Freeware (свободное программное обеспечение)*, где free происходит от слова свободный, а не бесплатный. Это программное обеспечение дает разработчикам и пользователям полную свободу и контроль над самым последним битом, работающим на их компьютере.

Но Python распространяется не по классической Freeware-лицензии *GPL (GNU Public License, публичная лицензия GNU)*, разработанной FSF *Free Software Foundation (Фонд свободного программного обеспечения)*. Лицензия GPL направлена на достижение цели FSF: свободное ПО для всего и всех, поэтому продукты, напрямую содержащие в себе код, предоставленный по лицензии GPL, обязаны также быть под GPL. Это обстоятельство по душе не всем производителям программ, которые, как правило, преследуют более узкие цели (например получение денег).

Python всегда распространялся под своей лицензией, похожей на лицензию на систему BSD, позволяющей производителям делать свои собственные изменения в коде и не открывать их, как того требует, например, лицензия GPL.

В настоящий момент права на интеллектуальную собственность, добавленную в Python после версии 2.0, принадлежат *PSF (Python Software Foundation, Фонд программного обеспечения Python)*. Второй пункт лицензии на Python 2.1.1 (см. файл `LICENSE`) гласит:

```
2. Subject to the terms and conditions of this License Agreement, PSF
hereby grants Licensee a nonexclusive, royalty-free, world-wide
license to reproduce, analyze, test, perform and/or display publicly,
prepare derivative works, distribute, and otherwise use Python 2.1.1
alone or in any derivative version, provided, however, that PSF's
License Agreement and PSF's notice of copyright, i.e., "Copyright (c)
2001 Python Software Foundation; All Rights Reserved" are retained in
Python 2.1.1 alone or in any derivative version prepared by Licensee.
```

Согласно условиям этого лицензионного соглашения, PSF предоставляет получателю лицензии неисключительные, свободные от оплаты, всемирные права воспроизводить, анализировать, тестировать, исполнять и/или показывать публично, готовить производные продукты, распространять или другим образом использовать Python 2.1.1 отдельно или совместно с производными версиями, если получатель лицензии сохранит за Python 2.1.1 отдельно или совместно с производными версиями лицензионное соглашение и упоминание об авторских правах "Copyright (c) 2001 Python Software Foundation; All Rights Reserved".

Таким образом, Python может использоваться для разработки любого программного обеспечения как коммерческого, так и свободного. Лицензия на Python 2.1.1 и Python 2.0.1 официально признана FSF как совместимая с "Open Source".

Следует особо подчеркнуть, что согласно лицензии на Python разработчики программы для него могут распространять свои программы на коммерческой, условно-бесплатной основе или как свободное программное обеспечение без каких-либо компенсаций владельцам авторских прав на Python. Это очень существенный плюс по сравнению с коммерческим программным обеспечением, где часто накладываются значительные ограничения на коммерческое распространение программ даже легальными пользователями транслятора языка программирования.

Для российских производителей программного обеспечения это означает лицензионную чистоту Python и возможность использовать его в проектах, ориентированных как на щепетильные западные рынки, так и на становление практики легального программного обеспечения в России.

7.3. Области применения

Уже сейчас Python наряду с Java и Perl широко используется в организации динамических Web-сайтов. Распространенный Web-сервер Apache имеет специальный модуль `mod_python` для встраивания сценариев на Python в HTML-страницы. Также возможно, что поддержка апплетов на Python будет включена в будущие версии Web-браузера Mozilla. Написанная на Python система Zope (аналог известной ColdFusion) свободно доступна и является удобным инструментом для организации совместной работы над содержанием.

Благодаря модулям для подключения к базам данных из Python-программ можно работать с Oracle, Interbase, PostgreSQL, MySQL и другими реляционными базами данных.

На основе Python можно строить непрерывно работающие системы реального времени. Для этого предусмотрена возможность перезагрузки модулей в работающую программу. Таким образом, систему можно обновлять без остановки. (Конечно, это требует большей аккуратности при программировании объектов и использовании встроенных средств). Такое использование возможно благодаря практическому отсутствию в интерпретаторе Python утечек памяти.

Python с успехом применяется и в научных исследованиях. Для ученых и специалистов очень важно иметь язык программирования высокого уровня с легкочитаемым синтаксисом и гибкими возможностями для обработки данных. Для эффективной работы над числовыми данными создан Numerical Python — расширение языка для работы с матрицами и другими структурами данных.

Успех Python в разработке приложений во многом объясняется наличием библиотек модулей. Эра Internet внесла свой отпечаток в их состав. На Python можно работать с сервисами Internet на разных уровнях: от высокоуровневых запросов к URL до работы с "сырыми" сокетами (raw sockets). Простейший Web-сервер поставляется в числе стандартных модулей Python, а пример Web-браузера (Grail) можно получить из сети Internet. Python дает отличную возможность создавать приложения, работающие в сети.

Python может найти применение также в прототипировании программного обеспечения, быстрой разработке приложений. Созданный таким образом прототип можно частично (только наиболее критичные к быстродействию части) или полностью (если настаивает заказчик) переписать на C/C++. Кроме того, в Python-программах можно использовать C/C++-библиотеки, написав модули-расширения. Это позволяет сочетать быстродействие программ на C и удобный синтаксис Python. Согласование библиотек можно почти полностью автоматизировать применением программы SWIG.

Существует версия Python, которая компилируется в байт-коды виртуальной Java-машины и прозрачно поддерживает Java-модули и Java-классы, давая разработчику выбор языка программирования на Java-платформе *(см. гл. 15)*. Python будет поддерживать и платформу .NET, если та получит широкое признание и распространение.

Еще одной немаловажной областью применения Python является обучение программированию и объектно-ориентированному программированию. Более того, одной из целей создания этого языка была возможность применения языка продвинутыми пользователями ПК. Программирующие пользователи могут не только быстро изучить Python, но и эффективно его использовать для автоматизации рутинных операций. По элегантности синтаксиса и продуманности концепции Python, пожалуй, превосходит популярный Visual Basic или Delphi. На основе Python можно организовать курс программирования и ООП в вузе. В отличие, например, от C++, объектно-ориентированное программирование в Python более прозрачно и позволяет без лишних синтаксических деталей разобраться в сути объектно-ориентированного подхода.

Другое современное веяние — XML — также не обошло Python стороной. Python имеет хороший набор инструментов для обработки XML-документов. Python 2.0 поддерживает Unicode и потому может с успехом применяться в многоязычном документообороте.

Активное ядро разработчиков языка быстро реагирует на появление новых технологий и создает необходимые модули для работы с ними. Развитие языка идет по схеме "Open Source", а все вносимые изменения обсуждаются с использованием процесса, подобного интернетовским RFC[2]: каждое вносимое предложение, называемое *PEP (Python Enhancement Proposal, предло-*

[2] RFC — request for comments — справка Internet. — *Ред.*

жение по расширению языка Python), получает уникальный номер и активно обсуждается в соответствующих списках рассылки и телеконференции **comp.lang.python**. Благодаря наличию исходных текстов любой программист может добавить в свой интерпретатор Python недостающую ему особенность, поделиться ею с другими или даже предложить в официальную версию языка. Например, существует версия Python для Palm OS.

Приведенный список не претендует на полноту. Python имеет довольно широкую сферу применения, хотя и не является универсальным средством для решения всех программных проблем. Основным ограничением применимости Python является не очень высокое быстродействие. Как шутливо заметил создатель языка — это единственный недостаток.

Автор надеется, что данная книга позволит расширить круг пользователей языка Python.

7.4. Где взять Python и программное обеспечение

В начале необходимо обзавестись дистрибутивом Python. Пусть это будет Python 2.1.1. Легче всего получить нужные файлы через Internet с Web-сайта **http://www.python.org**. Уже на первой странице вы увидите ссылки на свежие версии интерпретатора языка. Можно зайти сразу на сайт **ftp://ftp.python.org/pub/python** и уже дальше выбирать нужную вам платформу и версию.

Фирма ActiveState Tool Corp. выпускает свой дистрибутив Python под названием ActivePython. Официальная страница этого программного продукта: **http://www.activestate.com/Products/ActivePython/**. ActivePython поставляется для различных Windows- и Linux-систем. В отличие от Python, лицензия на дистрибутив ActivePython ограничивает его "персональным использованием" или использованием "в рамках одной организации" в любых целях, в том числе и в коммерческих. Это не очень большое ограничение, т. к. любой может взять ActivePython с сайта фирмы.

Можно найти интерпретатор Python и для менее распространенных систем. Например, Python для Macintosh может быть найден по адресу **http://www.cwi.nl/~jack/macpython.html**, а для OS/2 — по адресу **http://warped.cswnet.com/~jrush/python_os2/**. Не поленитесь заглянуть на **http://directory.google.com/Top/Computers/Programming/LanguagePython/Ports/**, где собраны ссылки на версии Python для различных систем.

Не менее важно найти нужные библиотеки и другое программное обеспечение для Python. К сожалению, пока что у Python нет центрального архива с модулями (по примеру CPAN для языка Perl), но с помощью поискового сервера (например, того же **http://google.com**) требуемая библиотека быстро найдется.

Если такая библиотека для Python отсутствует, но имеется подходящая библиотека на стандартном C/C++, можно попробовать самостоятельно соорудить привязки этой библиотеки к Python. Для облегчения этого процесса обычно используется *SWIG (Simplified Wrapper and Interface Generator, упрощенный упаковщик и генератор интерфейса)*, см. **http://www.swig.org**.

На прилагаемом к этой книге компакт-диске вы найдете как интерпретатор Python, так и некоторые дополнительные программные библиотеки, не входящие в стандартную поставку Python.

7.5. Установка Python

Python поддерживает все основные платформы современных компьютеров от UNIX до MacOS, но процедуры установки Python в этих системах могут значительно различаться. Далее мы рассмотрим процесс установки Python под Windows, Linux и UNIX.

Итак, вы обзавелись исполняемым файлом с установщиком Python, который обычно имеет имя `Python-версия.exe`. Его можно просто запустить. Установщик работает по обычной для Windows схеме: задает вопросы, на которые можно отвечать, а можно просто нажимать кнопку **Next** (Далее), чтобы перейти к следующему кадру, и так далее пока не появится экран, на котором нужно нажать кнопку **Finish** (Закончить). По умолчанию Python установит все компоненты — это как раз то, что нужно для новичка. В русской версии Windows относящиеся к Python программы будут доступны в меню **Программы | Python**.

Предупреждение

Если вы собираетесь применять Python для обучения и намерены использовать оболочку IDLE, возьмите Python 1.5.2, т. к. в более поздних версиях оболочка IDLE некорректно работает с кириллицей. Практически примеры из первой части книги будут работать в Python 1.5.2 и IDLE.

Чтобы Python можно было вызывать из командной строки, необходимо добавить в файл autoexec.bat следующую строку:

```
PATH c:\python21;%PATH%
```

Здесь `python21` — каталог, в который Python был установлен.

Если вы используете Linux (например, RedHat Linux), то скорее всего интерпретатор Python уже установлен в вашей системе (попробуйте набрать `python` в командной строке). Если нет, то все будет зависеть от того, какой у вас дистрибутив. В основанных на RPM дистрибутивах вы можете установить Python командой:

```
rpm -i путь1 путь2 ...
```

где под путями понимаются полные имена файлов или их адреса в сети. В последнем случае файл будет автоматически выгружен из сети. Никаких вопросов установщик задавать не будет. Выбор компонентов определяется выбором устанавливаемых файлов.

Конечно, нужно внимательно смотреть, для какого дистрибутива Linux предназначен RPM-пакет. Можно взять пакет с исходными кодами (`src.rpm`) и попробовать его скомпилировать, но обычно для этого должны быть установлены кое-какие дополнительные библиотеки, об отсутствии которых вы узнаете при сборке (компиляции) исходного кода.

Для любой системы UNIX подойдет установка Python из архива с исходными кодами. Такой архив как правило имеет имя `Python-версия.tar.gz` или просто `Python-версия.tgz`. Последовательность действий в этом случае следующая (пользователи UNIX ее хорошо знают):

```
tar -zxf Python-версия.tgz      # распаковываем архив
cd Python-версия                # переходим в каталог дистрибутива
./configure                     # конфигурируем (по умолчанию)
make                            # компилируем
make install                    # устанавливаем
```

7.6. Где получить помощь

Вполне возможно, что даже прочитав эту книгу от корки до корки, вы ощутите потребность получить конкретную помощь или узнать о Python и его расширениях больше. Если вы подключены к Internet, можно считать, что проблема решена, т. к. помощь по Python может быть оказана:

- в телеконференции **news:comp.lang.python**;
- в списках рассылки, посвященных Python, самый посещаемый из которых — **python-list@python.org**. Об остальных списках рассылки можно узнать по адресам **href:http://www.python.org/psa/MailingLists.html** и **http://aspn.activestate.com/ASPN/Mail/**;
- в специальных каналах irc, доступных на серверах **irc.openprojects.net** это канал **#python**;
- если вы напишите по адресу **help@python.org** (на английском языке), то вам поможет один из специалистов-добровольцев;
- на русском языке вам могут помочь: **http://sultan.virtualave.net**, **http://zope.net.ru**. Для того чтобы подписаться на русскоязычный список рассылки по Python и Zope, нужно отправить письмо на адрес **python-subscribe@itconnection.ru**.

Кроме того, существует несколько *групп по интересам (SIG, Special Interest Group)*, посвященных различным вопросам применения Python, см. **http://www.python.org/sigs**.

При обращении за помощью нужно четко сформулировать проблему, указав, в какой операционной системе вы работаете, с какой версией Python и версии используемых библиотек. Если проблема кроется в определенном фрагменте кода, лучше привести этот фрагмент (разумеется, убрав ненужные детали).

Наконец, существенно углубить свои знания о Python и приобщиться к примеру превосходно выполненного программирования можно, изучая исходные тексты интерпретатора Python и стандартных модулей. Исходные тексты свободно доступны в сети Internet.

Если вам необходимо узнать, какие функции и переменные предоставляет некоторый модуль, можно попробовать использовать встроенную функцию `dir()`. В примере ниже мы изучаем модуль `statvfs` (здесь использован Python 2.1):

```
>>> import statvfs
>>> print dir(statvfs)
['F_BAVAIL', 'F_BFREE', 'F_BLOCKS', 'F_BSIZE', 'F_FAVAIL', 'F_FFREE',
'F_FILES', 'F_FLAG', 'F_FRSIZE', 'F_NAMEMAX', '__builtins__',
'__doc__', '__file__', '__name__']
>>> print statvfs.__doc__
Constants for interpreting the results of os.statvfs() and os.fstatvfs().
```

Из чего мы можем сделать вывод, что `statvfs` предоставляет константы для интерпретации результатов функций `os.statvfs()` и `os.fstatvfs()`.

В Python 2.1 есть еще более удобный механизм получения помощи — модуль `pydoc` *(см. разд. 9.1)*.

Глава 8



Описание языка

В этой главе описаны синтаксис и семантика типов данных, операций, конструкций и операторов языка Python; функции, специальные методы классов и исключения, а также некоторые другие вопросы, относящиеся непосредственно к языку.

8.1. Структура программы

Программа на языке Python может быть передана на вход интерпретатора несколькими способами.

Самый распространенный вариант — *файловый ввод*. В этом случае программа — последовательность строк с операторами и пустых строк. Этот вариант используется при синтаксическом анализе цельной Python-программы из файла, анализе модуля или анализе аргумента оператора `exec`. Например:

Листинг 8.1. Файл example.py. Пример простейшей программы

```
#!/usr/bin/python

print "2 * 2 =", 2*2
print "2 * 3 =", 2*3
```

Запустить программу можно с помощью команды `python example.py`.

> **Примечание**
>
> Первая строка данного файла соответствует принятому в UNIX соглашению об исполняемых файлах. Она содержит имя программы, которую нужно запустить для интерпретации данного файла. Пометить файл как исполняемый можно с помощью команды:
>
> ```
> chmod +x example.py
> ```
>
> После чего ее можно запускать саму по себе:
>
> ```
> ./example.py
> ```

Второй способ — *интерактивный (диалоговый) ввод программы*. Он отличается тем, что после составных операторов должен быть дополнительный перевод строки. Например:

Листинг 8.2. Диалог с интерпретатором Python

```
>>> for i in [1,2,3]:
...   print i**2,
...
1 4 9
>>>
```

где '>>>' и '...' — приглашения интерпретатора ко вводу.

Третий способ — *ввод выражений*. Он употребляется в функции `eval()` и при применении функции `input()`. И в том, и в другом случае выражения Python задаются через запятую. Например:

```
>>> input()
2*2,3*3
(4, 9)
>>> print eval("2+2,5*6")
(4, 30)
```

8.1.1. Физические и логические строки программы

Файл с исходными текстами программ на языке программирования является текстовым файлом, принятым на данной компьютерной платформе. Под *текстовым файлом* обычно понимается файл, состоящий из букв, цифр, пробелов, знаков пунктуации и других символов с кодами ASCII от 32 до 255, разбитых на строки с помощью признаков концов строк (в UNIX — код 13, на Macintosh — 10, в DOS/Windows — 10 и 13, идущие подряд). Конечно, в текстах иногда используются *символы табуляции* (код 9) и, много реже, другие управляющие символы. Такие строки будем называть *физическими*.

Логические строки состоят из нескольких физических строк. В языке Python для этого есть явные и неявные правила. Оператор Python не может выходить за пределы логической строки (за исключением операторов, состоящих из нескольких частей, таких как `try-except`), а писать очень длинные строки неудобно и некрасиво.

Для явного разбиения логической строки на несколько физических можно использовать обратную косую черту "\".

Неявное разбиение может происходить внутри строковых литералов, записанных с помощью утроенных кавычек или апострофов, а также внутри любого вида скобок: "()", "[]" или "{}".

Следующий пример иллюстрирует различные варианты записи операторов на нескольких строках:

```
print "Один оператор -- одна логическая строка"
try:
  print "Оператор try-except"
except: print "занимает две строки"
if 2 > 1  \
   and 3 > 2 \
   and 4 > 3:
  print "Сложные условия легче воспринимать так."
if (2 > 1
   and 3 > 2
   and 4 > 3):
  print "Внутри скобок не нужен \\"
print "Словарь может занимать несколько строк", {
   1 : "one",
   2 : "two",
   3 : "three",
}[2]
print """Строковая константа \
в утроенных кавычках может
занимать несколько строк как с явным разбиением,
так и без него."""
```

Результат выполнения:

```
Один оператор -- одна логическая строка
Оператор try-except
Сложные условия легче воспринимать так.
Внутри скобок не нужен \
Словарь может занимать несколько строк two
Строковая константа в утроенных кавычках может
занимать несколько строк как с явным разбиением,
так и без него.
```

8.1.2. Комментарии и пустые строки

Комментарии являются важной частью любой программы. Они позволяют оставить весточку для тех, кто будет в дальнейшем разбираться в написанной программе, даже если это будет ее автор. К сожалению, не все понима-

ют важность написания комментариев, жалеют на них времени. О пользе комментариев *см. разд. 8.10*.

Комментарии записываются после *символа "#"* (решетка или диез) и продолжаются до конца физической строки. Комментарии не могут быть помещены внутрь строковой константы (символ "#" будет воспринят сам по себе), а также после обратной косой черты "\" (слэш), обозначающей разбиение логической строки на физические. Сами же комментарии не могут продолжаться на следующей строке при помощи обратного слэша. Комментарии игнорируются интерпретатором языка. Именно поэтому комментарии можно использовать для временного исключения части кода из программы.

В системе UNIX почти все сценарные языки используют "#" для обозначения комментариев, что нашло свое отражение в применении первой строки для обозначения того, какой интерпретатор должен выполнять записанную далее программу. (В UNIX любой текстовый файл может быть исполнен наравне с бинарной программой, если задать интерпретатор и установить флаг исполняемости.)

Пустые строки состоят из *пробельных символов* (пробелов, табуляций, переводов страниц). Помимо этого, в строке может находиться комментарий. Пустые строки игнорируются интерпретатором Python (если, конечно, они не относятся к многострочным строковым константам). При интерактивном использовании интерпретатора *(см. разд. 8.1)* полностью пустые строки нулевой длины служат для обозначения окончания составных операторов.

Листинг 8.3. Файл rem.py. Использование комментариев, строк документации и пустых строк в программе

```
#!/usr/bin/python
# В предыдущей строке указано, где именно лежит
# интерпретатор Python в UNIX

"""А это -- строка документации. Ее иногда используют
для многострочных комментариев, что удобно, но строго
говоря неправильно."""

# предыдущая строка была пустой, она игнорируется
print """Здесь пустая строка

имеет значение.     # это не комментарий
"""
a = [
```

```
  1,          # в неявно продолженной строке можно писать комментарии
  2,
  3,
]
```

Результат выполнения:

```
Здесь пустая строка

имеет значение.     # это не комментарий
```

8.1.3. Отступы и пробелы

Отступы (indents) в Python играют очень важную синтаксическую роль. От их применения зависит группировка операторов в составные операторы, а, значит, структура программы и логика ее работы. В других языках программирования отступы чаще всего являются элементами стиля, в Python — это требование синтаксиса, делающее программы читаемыми не только людьми, но и интерпретатором языка.

Для создания отступов могут использоваться не только пробелы, но и *табуляции*. Последние приводятся к соответствующему числу пробелов самим интерпретатором. В пишущих машинках табуляции служили для отсчета фиксированных отступов, следующих обычно через 8 позиций. При нажатии на табуляцию каретка такой машинки пропускала от 1 до 8 знакомест, смещаясь вправо до следующей позиции табулятора. Интерпретатор Python поступает похожим образом, заменяя символ табуляции на 1—8 пробелов до следующей позиции с номером, кратным 8. Символы перехода на новую страницу (formfeed) в начале строки игнорируются, что позволяет разбить листинг программы на страницы для печати, не нарушая логики программы. Получаемое общее число пробелов до первого непробельного символа и составляет значение отступа для данной логической строки.

Здесь стоит отметить, что лучше придерживаться какого-то одного способа при создании отступов (либо пробелы, либо табуляции). Автор этой книги использует для отступов два пробела, т. к. больший отступ "отъедает" от строки слишком много места, а меньший плохо различим на глаз. Кроме того, используемый автором текстовый редактор Emacs умеет автоматически менять величину отступа по всему файлу с программой.

Операторы Python могут содержать в себе вложенные фрагменты кода. Величина отступа в рамках одного фрагмента должна быть одинакова. Чем больше уровень вложенности операторов — тем больше должна быть величина отступа. По окончании фрагмента (или фрагментов) кода величина отступа должна возвратиться к значению, использованному до начала этого фрагмента. На самом внешнем уровне отступ должен быть нулевым. В стро-

ках, в которых нет ничего, кроме комментариев (заданных с помощью "#"), а также у продолжений логических строк отступы могут быть любыми. Другими словами, отступы важны лишь в начале логических строк.

```
def sum(lst):
  """Сумма элементов списков -- целых чисел"""
  s = 0
#  Суммируем только списки
  if type(lst) != type([]):
     lst = []
  k = 0
  for i in lst:
       if type(i) == type(1):
         s = s + i
         k = k + 1
  print "Просуммировано элементов: %i" % k
  return s

print "Сумма:"
print sum([1, "a", 23, [1, 2, 3]])
```

Результат выполнения:

```
Сумма: Просуммировано элементов: 2
24
```

Приведенный пример использует отступы различной величины. Это не очень красиво, но приемлемо для интерпретатора Python. Следующий пример демонстрирует ошибки при использовании отступов:

```
def sum(lst):             # здесь должен быть нулевой отступ
    """Сумма элементов списков -- целых чисел"""
s = 0                      # здесь не сделано должного отступа
#  Суммируем только списки
    if type(lst) != type([]):
     lst = []
    k = 0
    for i in lst:
       if type(i) == type(1):
         s = s + i
       k = k + 1           # по логике программы здесь должен
                           # быть отступ наравне с предыдущим
```

```
    print "Просуммировано элементов: %i" % k
      return s              # ошибка: неожиданный отступ

  print "Сумма:"            # некорректно, т. к. не
                            # соответствует никакому уровню
print sum([1, "a", 23, [1, 2, 3]])
```

8.2. Типы данных и объекты

Все данные в Python-системе представлены в виде объектов и связей между объектами. Каждый объект обладает *индивидуальностью* (то, что отличает его от других объектов), *значением* (то, что может меняться в объекте, его состояние) и *типом* (см. ниже).

Индивидуальность (identity) — это то, что может различаться, даже если тип и значение совпадают. Для простейших встроенных типов (числа, строки) индивидуальность определяется значением. Индивидуальность дается объекту при "рождении". Операция `is` позволяет сравнить идентичность (совпадение индивидуальностей) двух одновременно находящихся в памяти объектов. Встроенная функция `id()` дает уникальное для каждого объекта число (в текущей реализации Python — адрес в оперативной памяти).

Значение (value) определяет текущее состояние объекта. Состояние некоторых объектов не меняется после их создания (неизменяемые объекты), состояние других может измениться. Конечно, это не совсем строго. Например, значение кортежа может измениться, если изменится содержащийся в нем объект (при этом, конечно, индивидуальность содержащихся в кортеже объектов измениться не может.) *Изменяемость (mutability) объекта* зависит от типа. Числа, строки, кортежи — *неизменяемые объекты (immutable)*, тогда как списки и словари — *изменяемые (mutable)*. Сравнить значения двух объектов можно с помощью операции `"=="` или ее отрицания `"!="` (`"<>"`).

Тип (type) определяет поведение объекта: изменение его состояний, допустимые над данным объектом операции. Традиционно под типом данных понимается множество значений, которые может принимать величина данного типа, со множеством определенных для данного типа операций. От типа зависит объем памяти для хранения данной величины и другие характеристики. Помимо использования встроенных типов данных программист может создать свои, определив *класс (class)*. Величины этого типа называются *экземплярами класса (class instances)* или *объектами (objects) (см. разд. 8.2.6)*. Язык Python не является строго типизированным языком, как Pascal, C или Java, в которых нужно явно описывать типы переменных, аргументов и результатов функций, атрибуты классов. В Python тип данных проверяется не на этапе компиляции в байт-код, а при исполнении соответствующей опе-

рации. Тип может изменяться — для Python характерно динамическое управление типами данных. Это получается потому, что все величины в Python являются объектами, а потому применимость операции определяется присутствием или отсутствием метода, соответствующего производимой операции. Такой подход имеет свои достоинства и недостатки.

Объект может иметь одно или несколько имен в одном или нескольких пространствах имен. В отличие от языка C, имена даются не выделенным областям памяти, а лишь связываются (возможно, на время) с некоторыми объектами в момент исполнения — позднее динамическое связывание Python. Даже после исполнения простейшего оператора `a=a+1` имя `a` будет связано уже с другим объектом.

```
lst = [1,2,3,4,5]          # заводим список
print "lst:", lst
print "Узнаем типы объектов:", type(125), type("строка"), type(lst)
lst1 = lst                 # даем списку еще одно имя
print lst is lst1, " -- lst и lst1 один и тот же объект"
print lst is [1,2,3,4,5], " -- а эти два объекта -- разные,"
print lst == [1,2,3,4,5], " -- несмотря на совпадение значений"
lst1[0] = 0                # изменяем первый элемент списка lst1
print "lst тоже изменился: ", lst
del lst                    # список не удалился, только его имя
del lst1                   # с удалением последнего имени
                           # список тоже будет удален
```

Результат выполнения:

```
lst: [1, 2, 3, 4, 5]
Узнаем типы объектов: <type 'int'> <type 'string'> <type 'list'>
1  -- lst и lst1 один и тот же объект
0  -- а эти два объекта -- разные,
1  -- несмотря на совпадение значений
lst тоже изменился:  [0, 2, 3, 4, 5]
```

Интерпретатор Python ведет счет ссылкам на объект. Объект существует, пока на него ссылаются, и не может быть уничтожен явной командой. (Команда `del` не уничтожает объект, а лишь убирает одну из ссылок на него). Объект, на который нет ссылок, удаляется из оперативной памяти. Несколько объектов могут быть недоступны из программы, но занимать память из-за того, что они ссылаются друг на друга. Последние версии Python имеют механизм по *сборке мусора (garbage collection)*, который находит такие объекты и освобождает из-под них память.

Замечание

Для записи объектов в долговременную память можно использовать модуль `shelve` *(см. разд. 9.2.10)*. При таком сохранении индивидуальность (`id`) не сохраняется.

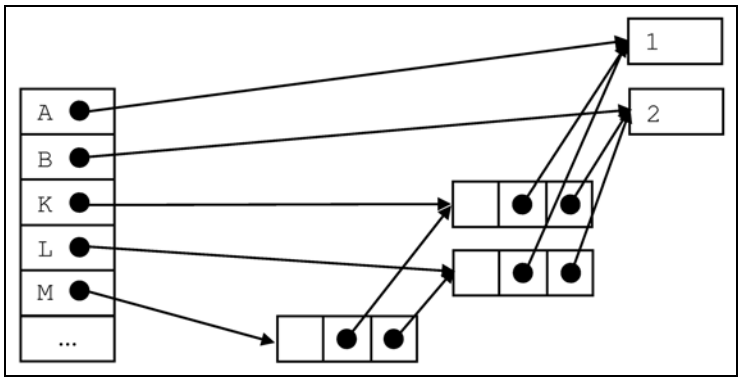


Рис. 8.1. Пространства имен, объекты и ссылки

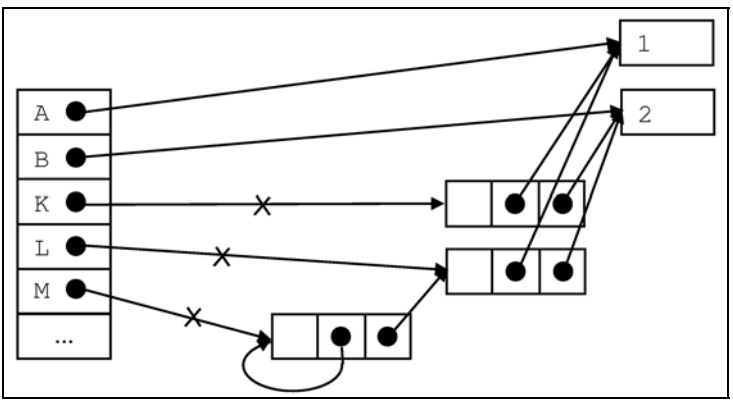


Рис. 8.2. Удаление объектов. Циклические ссылки

На рис. 8.1 и 8.2 проиллюстрированы пространства имен и происходящие в них изменения при удалении имени для следующего примера:

```
A = 1
B = 2
K = [A, B]
L = [A, B]
M = [K, L]
```

```
print globals()
# см. рисунок 8.1
M[0] = M        # создаем циклическую ссылку
del K, L, M     # удаляем объекты
# см. рисунок 8.2
```

Оператор `del` удаляет ссылки имен на объекты (обозначено крестиками). Из памяти же объект удаляется только при отсутствии на него ссылок. Таким образом, объект `K` будет удален из памяти, а объект `M` и вместе с ним объект `L` интерпретатор убрать не сможет — они так и останутся в памяти, пока не завершится интерпретатор Python или память из-под объектов `L` и `M` не будет освобождена процедурой сборки мусора.

8.2.1. Типы *None* и *Ellipsis*

Особое место в Python занимают типы `None` (ничто) и `Ellipsis` (многоточие).

Тип `None` имеет единственное значение `None` и единственный объект с таким значением. Для доступа к объекту используется встроенное имя `None`. Значение `None` используется для обозначения отсутствия значения. Например, функция, не имеющая `return`, возвращает `None`. Логическое значение `None` — ложь.

Тип `Ellipsis` тоже имеет единственное значение `Ellipsis` и единственный объект с таким значением, доступный по имени `Ellipsis`. Значение используется как `'...'` в сложных многомерных срезах *(см. гл. 11)* для указания произвольного числа измерений. Логическое значение `Ellipsis` — истина.

```
>>> from Numeric import *
>>> a=array(range(25))
>>> a.shape=[5,5]
>>> a[...,1:2]
array([[ 1],
       [ 6],
       [11],
       [16],
       [21]])
```

8.2.2. Числа

Практически в любом языке программирования, от машинных кодов до языков сверхвысокого уровня, числа описываются одним или несколькими типами данных. Как правило, это целые числа и числа с плавающей точкой

разной точности. Python имеет четыре числовых типа: целые (`int`), длинные (произвольной точности) целые (`long`), числа с плавающей точкой (`float`) и комплексные числа (`complex`). Представление соответствующих чисел зависит от используемой платформы и компилятора C. О целых числах можно сказать, что для их представления требуется не менее 32 битов (используются длинные целые языка C), для представления чисел с плавающей запятой используется тип `double` (компилятора C). Комплексные числа составлены из двух чисел с плавающей точкой, а длинные целые в Python могут быть очень длинными: насколько хватит памяти. В табл. 8.1 приведены ограничения для различных типов чисел.

Таблица 8.1. *Диапазоны представления чисел*

Тип числа	Диапазон значений
Целое	От –2 147 483 647 до +2 147 483 647 (на 32-разрядной платформе)
Длинное целое	Произвольное, ограничено оперативной памятью
Число с плавающей точкой	От 4.9406564584124654e-324 до 1.7976931348623157e+308 (на 32-разрядной платформе)

Числа вводятся в выражения на Python либо в виде констант, либо как результаты операций. Запись целых и чисел с плавающей точкой стандартна для современных языков программирования. Целые числа могут быть заданы не только в десятичной системе счисления, но и в восьмеричной (префикс "`0`") и шестнадцатеричной (префикс "`0x`"). Добавлением "`L`" или "`j`" (в верхнем или нижнем регистре) задаются длинные целые и комплексные числа. Например:

```
123
010
0xFF
10000000000000000000000000L
12.4
2.3E-12
1j
1+2.5j
```

Отрицательные числа получаются в результате операции смены знака — унарный минус — над соответствующим положительным числом, а комплексные числа — в результате сложения констант мнимого и действительного чисел. Подробнее о типах констант *см. статью "Константы (литералы)" в разд. 8.3.1.*

Python позволяет смешивать в одном арифметическом выражении числа разных типов. Различные типы будут при проведении бинарных операций преобразованы к общему типу. Целые приводятся к длинным целым, те — к числам с плавающей точкой, последние — к комплексным числам. При сравнении числа также приводятся к общему типу. В табл. 8.2 приведен список операций над числами.

Таблица 8.2. *Операции над числами*

Операция	Назначение
`x + y`	Сумма
`x - y`	Разность
`x * y`	Произведение
`x / y`	Частное от деления
`x % y`	Остаток от деления
`-x`	Смена знака
`+x`	`x` остается неизменным
`abs(x)`	Абсолютное значение
`int(x)`	Преобразование к целому
`long(x)`	Преобразование к длинному целому
`float(x)`	Преобразование к числу с плавающей точкой
`complex(x,y)`	Преобразование к комплексному числу `x+yj`, по умолчанию `y==0`
`z.conjugate()`	Сопряженное комплексному числу `z`
`divmod(x, y)`	Пара `(x / y, x % y)`
`pow(x, y)` или `x**y`	Возведение `x` в степень `y`

Как и в языке C, необходимо осторожно обращаться с делением чисел. Например, выражение `12/45*45` дает неожиданный результат: `0`. Это происходит потому, что при делении целых чисел не возникают числа с плавающей точкой, а берется только целая часть. Для числовых констант достаточно поставить точку, чтобы недоразумение исчезло: `12./45*45 == 12.0`. В выражениях с целыми переменными можно указать явное преобразование типа: `float(a)/b*c`. *(См. статью "Функции преобразования типов" в разд. 8.6.1.)*

Примечание

В версиях Python 2.2 и старше для целочисленного деления будет применяться другая операция (скорее всего, `"//"`). При этом старая операция `"/"` в одной из будущих версий будет реализовывать более естественное деление.

8.2.3. Последовательности

Python имеет шесть типов последовательностей: строки, Unicode-строки, списки, кортежи, буферы и xrange-объекты. Первые четыре типа описаны отдельно в последующих разделах, а здесь будут перечислены операции и функции, общие для всех последовательностей.

В табл. 8.3 приведены операции, свойственные последовательностям. Из всех типов последовательностей только списки являются изменяемыми последовательностями и имеют дополнительный набор операций (табл. 8.5).

Таблица 8.3. *Операции над последовательностями*

Операция	Результат	Примечание
`len(s)`	Длина последовательности `s`	
`x in s`	1 если в последовательности `s` есть элемент, равный `x`, иначе 0	
`x not in s`	Логическое отрицание `x in s`	
`s + s1`	Конкатенация последовательностей	
`s * n , n * s`	Последовательность из `n` раз повторенной `s`	Если `n` меньше 0, возвращается пустая последовательность
`s[i]`	`i`-й элемент `s`, считая с 0	Если `i` или `j` отрицательны, то индекс вычисляется относительно конца последовательности. Например, вместо отрицательного `j` берется `len(s)+j`. Таким образом, `s[-1]` позволяет взять последний элемент последовательности
`s[i:j]`	Срез из последовательности `s` от `i` до `j`	Семантика срезов описана в *статье "Объекты-срезы" в разд. 8.2.10*. Здесь можно лишь напомнить, что `s[i:j] == [s[i], s[i+1], ..., s[j-1]]`
`min(s)`	Наименьший элемент `s`	
`max(s)`	Наибольший элемент `s`	

Так как единственным представителем изменяемой последовательности является список, операции, свойственные таким последовательностям, описаны в *статье "Списки" этого раздела.*

Следует также отметить, что определенные пользователем классы могут эмулировать последовательности, если будут иметь определенный набор методов *(см. разд. 8.4.4)*.

Объекты типа `xrange` используются в цикле `for` для задания (эмулированной) последовательности, которую будет пробегать параметр цикла. Функция `xrange()` аналогична функции `range()` с той разницей, что `range()` формирует в памяти (возможно, очень длинный) список, а `xrange()` порождает объект `xrange`, последовательно выдающий значения для цикла `for`.

```
s = 0L
for i in xrange(10001):
   s += i
print s
```

Результат выполнения:

```
50005000
```

Объекты `xrange` лучше не применять в качестве полноправных последовательностей, ограничившись взятием i-го элемента, длины последовательности и, конечно, использованием в цикле `for`.

Строки

Строка (string) — последовательность символов. В языке Python строка является неизменяемой структурой данных: созданную строку нельзя изменять. Можно только создать новую строку, скопировав часть элементов старой строки. Поэтому операции со строками приводят к операциям копирования строки. В отличие от списка или кортежа, но аналогично числовым типам данных, операция присваивания приводит к копированию строки, как это принято в других языках программирования. К строкам применимы операции над последовательностями и другие, чисто строковые операции.

Результат выполнения:

```
Первый пример - "строка в кавычках"
Второй пример - "кавычки" внутри апострофов
```

Внутри утроенных кавычек (или апострофов) можно писать и ", и ' и даже не обращать внимания на переводы строк. В таких строках удобно размещать код HTML, например:

```
<a href="http://www.python.org">Python</a>
```

Нам не пришлось писать так:

```
<a href="http://www.python.org">Python</a>
Строки в обычных кавычках тоже можно размещать в
```

```
нескольких программных строках. Но для этого приходится
явно расставлять признаки перевода строки (\n),
если они нужны.
```

Это пример сырой строки. Здесь символ `\` не имеет своего обычного смысла и `\n` будет введена в строку буквально.

В Python 2.0 у строк появились многочисленные атрибуты. Все они имеют функции-аналоги в модуле `string`. В табл. 8.4 приведен список методов, которые есть у строк. Эти методы применимы даже к строковым константам.

Таблица 8.4. *Методы строк и Unicode-строк*

Метод и его аргументы	Описание
`s.capitalize() -> s1`	Делает первое слово с прописной буквы
`s.center(w) -> s1`	Центрирует строку в поле длины `w`
`s.count(sub[, start [, end]]) -> i`	Число вхождений строки `sub` в строке `s[start:end]`
`s.encode([encoding [,errors]]) -> s1`	Возвращает строку в заданной кодировке `encoding`. Параметр `errors` может быть `"strict"` (умолчание), `"ignore"`, `"replace"`
`s.endswith(suffix [, start[, end]]) -> 0, 1`	Оканчивается ли `s` на suffix. Если задан `start`, то проверяется `s[start:]`, если задан и `end`, то — `s[start:end]`
`s.expandtabs([tabsize]) -> s1`	Заменяет знаки табуляции на пробелы. По умолчанию величина табуляции `tabsize` равна `8`
`s.find(sub [,start [,end]]) -> i`	Возвращает наименьший индекс, с которого начинается вхождение подстроки `sub` в строку `s`. Как обычно, `start` и `end` ограничивают поиск окном `start:end`, но индекс выдается по исходной строке. Если подстрока не найдена, возвращается `-1`
`s.index(sub [,start [,end]]) -> i`	Аналогично `find()`, но возбуждает `ValueError` в случае неудачи
`s.isalnum() -> 0, 1`	Возвращает `1`, если `s` содержит только буквы и цифры и имеет ненулевую длину
`s.isalpha() -> 0, 1`	`1`, если `s` содержит только буквы и длина `S` ненулевая
`s.isdecimal() -> 0, 1`	`1`, если `s` содержит только десятичные знаки (только для строк Unicode) и длина не ноль
`s.isdigit() -> 0, 1`	`1`, если содержит только цифры и длина ненулевая
`s.islower() -> 0, 1`	Возвращает `1`, если все буквы строчные (и их более одной), иначе — `0`

Таблица 8.4 *(продолжение)*

Метод и его аргументы	Описание
`s.isnumeric() -> 0, 1`	`1`, если в строке только числовые знаки. (Только для Unicode)
`s.isspace() -> 0, 1`	Возвращает `1`, если строка состоит только из пробельных символов. Внимание! Для пустой строки возвращается `0`
`s.istitle() -> 0, 1`	`1`, если все слова строки начинаются с большой буквы и продолжаются строчными буквами, примерно соответствуя английским правилам написания заголовков
`s.isupper() -> 0, 1`	Метод возвращает `1`, если в строке все буквы прописные и есть хотя бы одна буква
`s.join(seq) -> s1`	Аналогично `string.join(seq, s)`. То есть соединение строк из `seq` через разделитель `s`
`s.ljust(w) -> s1`	Возвращает строку, выровненную по левому краю с помощью пробелов в поле ширины `w`
`s.lower() -> s1`	Приводит строку к нижнему регистру букв
`s.lstrip() -> s1`	Удаляет пробельные символы слева
`s.replace (old, new[, n]) -> s1`	Возвращает копию строки `s`, в которой подстроки `old` заменены на `new`. Если задано `n` — то только заменяются первые `n` вхождений
`s.rfind(sub[, start [,end]]) -> i`	Возвращает наибольший индекс, начиная с которого строка `sub` располагается в строке `s`. Если задан `start`, то поиск происходит в `s[start:]`, а если есть и `end`, то в `s[start:end]`. Индекс все равно соответствует исходной строке. Если подстрока в строке не найдена, возвращается `-1`
`s.rindex(sub[, start [,end]]) -> i`	Аналогично `rfind()`, но в случае неудачи возбуждается исключение `ValueError`
`s.rjust(w) -> s`	Выравнивание по правому краю в поле шириной `w`. При необходимости, строка дополняется слева пробелами
`s.rstrip() -> s`	Удаляет пробельные символы справа
`s.split([sep[, n]]) -> list`	Возвращает список подстрок, получающихся разбиением строки `a` разделителем `sep`. Параметр `n` задает максимальное количество разбиений
`s.splitlines ([keepends]) -> list`	Возвращает список строк (то есть, разбиение происходит по признакам окончания строк). Если `keepends` — логическая истина, то сохранить признаки концов строк в списке-результате

Таблица 8.4 *(окончание)*

Метод и его аргументы	Описание
`s.startswith(prefix [, start[, end]]) -> 1, 0`	Аналогично `endswith()`, но проверяет, начинается ли строка `s` с подстроки `prefix`. Если заданы `start` и `end`, то проверяется только `s[start:end]`
`s.strip() -> s1`	Удаляет пробельные символы с обоих краев
`s.swapcase() -> s`	Меняет регистр букв на обратный
`s.title() -> s`	Возвращает строку, в которой все слова начинаются с большой буквы
`u.translate(table) -> u1`	Производит преобразование с помощью таблицы перекодировки `table` — словаря для перевода кодов в коды (или в `None`, в случае чего соответствующий символ просто удаляется)
`s.translate(table [, deletechars]) -> s1`	То же, но для обычных строк. Вместо словаря — строка перекодировки на 256 символов (ее можно сделать с помощью функции `string.maketrans()`). Необязательный параметр `deletechars` задает строку с символами, которые подлежат удалению
`s.upper() -> s1`	Переводит буквы строки в верхний регистр

Примечание

Результаты некоторых методов зависят от текущих установок "культурной среды" (locale) *(см. разд. 9.21.1)*.

Следующий пример посвящен различным видам выравнивания:

```
>>> a = "   строка для примера   "
>>> print a
   строка для примера
>>> print a.strip().center(30)
      строка для примера
>>> print a.strip().rjust(30)
            строка для примера
>>> print a.strip().ljust(30)
строка для примера
```

Альтернативой использованию методов для работы со строками является использование модуля `string` *(см. разд. 9.3.1)*.

Кортежи

Кортеж (tuple) — последовательность из фиксированного числа разнородных элементов. В языке Python кортеж является неизменяемой структурой данных, т. к. элементы созданного кортежа нельзя изменять. Также нельзя добавлять элементы к существующему кортежу. (Добавлять-то можно, только результат будет уже другим кортежем.) Можно считать кортеж списком-константой. К кортежу применимы операции выделения отдельных элементов и слоев.

Кортежи, в основном, применяются для агрегирования данных в более сложные структуры, а также для обеспечения возврата нескольких значений из подпрограммы. Естественно применять кортежи для хранения координат точки в пространстве или обозначения цвета в RGB-формате. Возможность хранения разнородных элементов — то, что отличает кортеж от массива. Кортеж легко превратить в список и наоборот с помощью функций `list()` и `tuple()` соответственно. Если не предполается изменять элементы последовательности и ее состав, эффективнее использовать кортежи.

```
Применение в присваиваниях, аргументах и возвращаемых значениях
def show_2_tuple((a, b), c):
  """Первый аргумент сразу разбирается как кортеж из двух элементов"""
  print "show_2_tuple:", a, b, c

def return_tuple(a, b):
  """Упрощенная запись кортежа при возврате значения"""
  return a+1, b+1

a, b, c = 1, 2, 3        # это
(a, b, c) = (1, 2, 3)    # и это -- одно и то же.
show_2_tuple((a, b), c)
print "return_tuple:", return_tuple(4, 5)
```

Результат выполнения:

```
show_2_tuple: 1 2 3
return_tuple: (5, 6)
```

Так как кортежи — неизменяемые объекты, их можно использовать в качестве ключей в словарях. Следующий пример показывает, как хранить разреженную матрицу в словаре, ключами к которому являются кортежи из координат:

```
# описание матрицы
sparse_matrix = {(0, 0): 1, (3, 3): 2, (3, 4): 1, (5, 4): -1}
# вывод матрицы
```

```
for i in range(6):
  for j in range(6):
    if sparse_matrix.has_key((i, j)):
      print sparse_matrix[i, j],
    else:
      print 0,
  print
```

Результат выполнения:

```
1 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 2 1 0
0 0 0 0 0 0
0 0 0 0 -1 0
```

Списки

Список (list) — последовательность разнородных элементов. Список является изменяемой структурой данных. Его можно наращивать, добавляя элементы методом `append()`, удалять элементы оператором `del`, заменять один или несколько элементов, сортировать элементы методом `sort()` или изменять порядок следования элементов на обратный методом `reverse()` и т. д.

Списки имеют широкое применение и большой набор встроенных и библиотечных функций для их обработки (в этом Python чем-то сродни языку LISP). Стиль Python работы со списками тяготеет к агрегированию операций над списками (вместо обработки в явно заданном цикле). Для этого в Python есть функции `map()` (отображение), `filter()` (фильтр), `reduce()` (сокращение), которые применяют действие функции (часто определенной тут же с помощью оператора `lambda`) ко всему списку. Такой подход не только уменьшает размер программы, но и позволяет увеличить ее производительность по сравнению с обычным циклом. Для обеспечения лучшей читаемости кода в этом случае можно разместить оператор на нескольких строках, например:

```
Пример обработки списка
# Справа указаны промежуточные результаты обработки списка:
p = reduce(lambda x,y: x+y,          # 3) суммируем элементы списка
           filter(lambda x: x>0,     # 2) берем положительные элементы
                  map(lambda x: x-1,# 1) отнимаем 1 от каждого элемента
                      range(-5,+5)  # [-5,-4,-3,-2,-1,0,1,2,3,4]
                      )             # [-6,-5,-4,-3,-2,-1,0,1,2,3]
```

```
                )                    # [1, 2, 3]
          )                          # 1+2+3 == 6
print p                              # 6
```

В отличие от массива, в списке можно хранить элементы разных типов, в том числе и списки. Список легко превратить в кортеж и наоборот с помощью функций `tuple()` и `list()` соответственно. Общее у строки, списка и кортежа то, что они являются последовательностями, поэтому строка легко становится списком, попадая на вход операций обработки списка или функции `list()`, например:

```
>>> print list("12345")
['1', '2', '3', '4', '5']
>>> print map(int, "12345")
[1, 2, 3, 4, 5]
```

Итак, списки являются встроенным типом данных Python, и к ним могут применяться операции, функции, методы и операторы, свойственные последовательностям *(см. разд. 8.2.3)*. То есть помимо операций из табл. 8.5 к спискам применимы операции из табл. 8.3.

Таблица 8.5. *Операции над списками*

Операция	Результат	Примечание
`s[i] = x`	`i`-й элемент списка `s` заменяется на `x`	
`s[i:j] = t`	Слой от `i` до `j` заменяется на (список) `t`	
`del s[i:j]`	Аналогично `s[i:j] = []`	
`s.append(x)`	Аналогично `s[len(s):len(s)] = [x]`	Добавляет элемент к списку
`s.count(x)`	Подсчитать количество `i`, для которых `s[i] == x`	
`s.extend(x)`	Аналогично `s[len(s):len(s)] = x`	Расширяет список
`s.index(x)`	Наименьшее `i`, такое что `s[i] == x`	Возбуждает исключение `ValueError`, если `x` не найден в `s`
`s.insert(i, x)`	Аналогично `s[i:i] = [x]` при `i >= 0`	
`s.pop([i])`	Аналогично `x = s[i]; del s[i]; return x`	По умолчанию `i=-1`

Таблица 8.5 *(окончание)*

Операция	Результат	Примечание
`s.remove(x)`	Аналогично `del s[s.index(x)]`	Возбуждает исключение `ValueError`, если `x` не найден в `s`
`s.reverse()`	Меняет порядок элементов `s` на обратный	
`s.sort ([cmpfunc])`	Сортирует элементы `s`. Можно задать свою функцию сравнения `cmpfunc`	См. примечание ниже

Примечание

Методы `reverse()` и `sort()` работают над списком на месте, не занимая дополнительной памяти под вспомогательный список. Методы возвращают `None`, чтобы напомнить об этом. Метод `sort()` позволяет пользоваться при сортировке другой функцией сравнения. Функция `cmpfunc(x, y)` должна возвращать `-1`, `0` или `1` в зависимости от результата сравнения: `-1` для "`x` больше `y`", `1` — "`x` меньше `y`" и `0` при равенстве. Стоит отметить, что отсортировать в обратном порядке быстрее с помощью `s.sort(); s.reverse()`, чем с использованием собственной функции сравнения.

Следующие несколько примеров показывают некоторые типичные приемы при работе со списками. Пример первый иллюстрирует некоторые операции и методы, описанные в табл. 8.5:

```
>>> l = range(6)                # порождаем список из 6 чисел от 0 до 5
>>> print l
[0, 1, 2, 3, 4, 5]
>>> l[1:1] = [0.5]              # вклиниваем еще одно значение после 0
>>> print l
[0, 0.5, 1, 2, 3, 4, 5]
>>> l[-1] = 5.01                # меняем последний элемент списка
>>> print l
[0, 0.5, 1, 2, 3, 4, 5.01]
>>> del l[-2]                   # удаляем предпоследний элемент
>>> print l
[0, 0.5, 1, 2, 3, 5.01]
>>> l[-1:] = [5, 5.1, 5.2, 2]  # вместо последнего элемента -- 4 других
>>> print l
[0, 0.5, 1, 2, 3, 5, 5.1, 5.2, 2]
>>> l.reverse()                 # переворачиваем список
```

```
>>> print l
[2, 5.2, 5.1, 5, 3, 2, 1, 0.5, 0]
>>> l.remove(2)                # удаляем первый элемент, равный 2, из
списка
>>> print l
[5.2, 5.1, 5, 3, 2, 1, 0.5, 0]
```

А следующий пример показывает, что список (в данном случае — список переменных) может стоять слева от присваивания:

```
[a, b, c] = [1, 2, 'c']
```

При обработке файлов журналов (log) часто возникает необходимость работать с отдельными полями журнала. Напрямую нельзя присвоить разбивку строки переменным, т. к. в случае несовпадения числа полей числу присваиваемых переменных возникнет ошибка. Пример в листинге 8.4 показывает, как поступить в случае, когда в разных строках файла журнала может быть разное число полей. Эффект достигается дополнением списка значений полей `record` значениями `None`, а затем взятием нужного числа элементов.

Листинг 8.4. Обработка файла журнала с переменным числом полей

```
import string
logfile = open("my.log", "rt")
while 1:
  record = string.split(logfile.readline())
  if not record:
    break
  t, user, address, inbytes, outbytes = (record + 5*[None])[:5]
  # Далее мы можем оперировать полученными переменными, но
  # переменные, которым не хватило значений, будут равны None
  print t, user, address, inbytes, outbytes
logfile.close()
```

В Python 2.0 появился новый мощный описательный элемент — списковые включения *(см. статью "Задание кортежей, списков, словарей" в разд. 8.3.1)*. В следующем примере списковое включение использовано для формирования списка пар из чисел 1, 2 и 3:

```
>>> L = [(x+1,y+1) for x in range(3) for y in range(3)]
>>> print "Пары:", L
Пары: [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2),
(3, 3)]
```

8.2.4. Словари

Словарь (dictionary) позволяет задавать соответствие (mapping) между значениями двух произвольных типов. Словарь — множество пар ключ—значение, где каждому *ключу (индексу)* соответствует только одно значение. Синтаксис записи словарей приведен в *статье "Задание кортежей, списков, словарей" в разд. 8.3.1.* Подобный словарю тип данных может называться *ассоциативным массивом* (как в LISP) или *хэшем* (как в Perl). В качестве ключей могут использоваться почти любые объекты. Изменяемые объекты не могут быть ключами. Обычно ключи — числа или строки, иногда кортежи.

Словари очень активно используются в программах на Python. Вот лишь некоторые, оформленные в синтаксисе и встроенных функциях языка применения:

- правый аргумент в операции заполнения строки по формату ("`%`") может быть словарем. В этом случае в строке формата можно задавать ключи, по которым будут подставляться соответствующие значения;
- результаты выполнения встроенных функций `locals()`, `globals()`, `vars()`;
- также с помощью словарей задаются пространства локальных и глобальных имен для функций `execfile()`, `eval()` и оператора `exec`;
- пространство имен модуля, класса или объекта также доступно в виде словаря (атрибут `__dict__`);
- формальные именованные аргументы доступны в теле определения функции в виде словаря.

```
def T(T_NAME, T_BODY, **attr):
  """создать тег HTML"""
  ATTS = ""
  for (a,v) in attr.items():
    ATTS += ' %(a)s="%(v)s"' % vars()
  return "<%(T_NAME)s%(ATTS)s>%(T_BODY)s</%(T_NAME)s>" % vars()

print T("a", "Главный сайт по Python", href="http://www.python.org")
```

Результат выполнения:

```
<a href="http://www.python.org">Главный сайт по Python</a>
```

Операции со словарями и методы, имеющиеся у словарей, даны в табл. 8.6.

Таблица 8.6. *Список операций со словарями*

Операция или метод	Результат	Примечания
`Len(s)`	Количество ключей в словаре	
`s.has_key(k)`	Есть ли в словаре ключ `k`	
`s[k]`	Значение, соответствующее ключу `k`	`KeyError`, если нет ключа
`s.get(k, x=None)`	`s[k]`, если есть `k`, иначе `x` (по умолчанию `x` равен `None`)	
`s[k] = x`	Значение, соответствующее ключу `k`, заменяется на `x`	
`s.setdefault (k, x=None)`	`s[k]`, если есть `k`, иначе `x` (или `None`, если `x` не задан). Также делает `s[k] = x`	
`del s[k]`	Удалить ключ `k` (и связанное с ним значение)	`KeyError`, если нет ключа
`s.update(s1)`	Добавляет к `s` ключи из `s1` (перезаписывая существующие)	
`s.items()`	Список пар (кортежей) ключ—значение	
`s.keys()`	Список ключей	
`s.values()`	Список значений (в том же порядке, что и ключи)	
`s.copy()`	Поверхностная копия словаря	
`s.clear()`	Очистка словаря от всех ключей (значений)	

Листинг 8.5. Примеры работы со словарями

```
>>> digits = {1 : 'one', 2 : 'two', 3 : 'three'}
>>> if digits.has_key(1):  # проверка наличия ключа в словаре
...   print digits[1]
one
>>> print digits.get(0, "zero")
zero
>>> print digits
{3: 'three', 2: 'two', 1: 'one'}
>>> print digits.setdefault(0, "zero")
zero
>>> print digits
```

```
{3: 'three', 2: 'two', 1: 'one', 0: 'zero'}
>>> print "items:", digits.items()
items: [(3, 'three'), (2, 'two'), (1, 'one'), (0, 'zero')]
>>> print "keys:", digits.keys()
keys: [3, 2, 1, 0]
>>> print "values:", digits.values()
values: ['three', 'two', 'one', 'zero']
>>> digits.update({3 : 'THREE', 4: 'FOUR'})
>>> digits[0] = 'ZERO'
>>> print digits
{4: 'FOUR', 3: 'THREE', 2: 'two', 1: 'one', 0: 'ZERO'}
```

В качестве ключей не могут использоваться списки и словари (и любые другие объекты, содержащие их), т. к. списки и словари являются изменяемыми объектами, а значение ключа не должно быть изменяемым объектом. Объекты, определенные пользователем, также можно использовать в качестве ключей, но, в отличие от встроенных типов, они будут сравниваться на идентичность, а не по значению, что в большинстве случаев — вовсе не то, что ожидалось.

```
class obj: # класс ради класса
  pass


class obj1: # класс чуть посложнее
  def __init__(self, x):
    self.value = x


o1 = obj1(0)
d = {o1 : 0, obj() : '1', obj1(1) : '2', obj1(2) : '3'}   # создаем
словарь
print d[o1]  # здесь все хорошо
print obj1(2)==obj1(2) # здесь что-то не ладно
d[obj1(1)]   # здесь ошибка: obj1(1) не тот же самый объект, что в ключе!
```

Результат выполнения:

```
0
0
Traceback (most recent call last):
  File "<stdin>", line 11, in ?
KeyError: <__main__.obj1 instance at 0x80fc214>
```

Словари реализованы как хэш-таблицы переменных размеров. Если вам необходимо, чтобы объекты некоторого класса могли служить ключами словаря, нужно дополнить класс методами `__hash__()` и `__cmp__()`. Первый будет давать *хэш-значение* объекта (например, отличное от `hash(obj)`), а второй позволит сравнивать два объекта (например, для разрешения коллизий в хэш-таблице). Для изменяемых объектов, у которых определен метод `__cmp__()`, нельзя определять еще и `__hash__()`, т. к. ключи словаря должны быть неизменяемыми объектами *(см. статью "Хэш-метод __hash__()" в разд. 8.4.1)*.

8.2.5. Типы с поддержкой вызова

Некоторые типы объектов Python поддерживают вызов подобно функциям. Назовем их типами с поддержкой вызова (`callable`). Сюда относятся функции (как встроенные, так и заданные пользователем), методы классов (встроенные и заданные пользователем), сами классы (вызовом класса создается новый объект данного класса), а также объекты определенных пользователем классов, имеющих метод `__call__()`.

Обычно вызываемым объектам передаются *параметры (аргументы)*. Эти параметры указываются в скобках после выражения с вызываемым объектом. Если никаких параметров не передается, скобки все равно ставятся.

```
print apply(int, ("123",)), callable(int)
funcs = {"i":int, "s":str, "f":float,}
for f in "isf":
  print f, funcs[f](12.3)
print vars()
```

Результат выполнения:

```
123 1
i 12
s 12.3
f 12.3
{'f': 'f', '__doc__': None, 'funcs': {'f': <built-in function float>,
's': <built-in function str>, 'i': <built-in function int>}, '__name__':
'__main__', '__builtins__': <module '__builtin__' (built-in)>}
```

Следующий пример предостерегает от распространенной ошибки, связанной с использованием атрибута вместо вызова метода:

```
class A:
  def check(x):
    return 0

a = A()
if a.check:   # ошибка! Правильно: a.check()
```

```
  print "проверка пройдена"
else:
  print "неудача"
```

Результат выполнения:

```
проверка пройдена
```

Сам по себе метод объекта является логической истиной, поэтому результат проверки прямо противоположен ожидаемому.

Ниже будут рассмотрены поддерживающие вызов объекты различных типов.

Функции

Любая функция сначала должна быть определена (в терминологии объектов — создана). *Определить функцию* можно с помощью операции `lambda` *(см. разд. 8.3.4)* или с помощью оператора `def` *(см. статью "Оператор def" в разд. 8.5.5)*. В результате возникает новый объект — функция. В языке Python единственной операцией, которую можно проделать над функцией, является *вызов (call)* с некоторыми *аргументами* (входными параметрами) и получение *результата* (выходного параметра). За исключением встроенных функций, определенных всегда, вызывать функцию можно только после *определения*. Здесь действуют те же правила, что и для других объектов *(см. разд. 8.7)*. В отличие от других языков программирования, в Python нельзя возвратить более одного выходного параметра, что совершенно не влияет на гибкость языка, т. к. возврат кортежа (или даже более сложной конструкции) позволяет вернуть много разных величин внутри одной последовательности. В описании функции аргументы называются *формальными*, а при вызове — *фактическими*.

Листинг 8.6. Определение и использование функций

```
print abs
def prev_month(year, month):
  # year и month -- формальные аргументы
  if month == 1:
     return (year-1, 12)
  return (year, month-1) # результат

f = prev_month
print prev_month(2001,4) # 2001 и 4 -- фактические аргументы
print f
print f(2002,1)
g = lambda year, month: [(year, month-1),(year-1, 12)][month==1]
```

```
print g
print g(2002,1)
```

Результат выполнения:

```
<built-in function abs>
(2001, 3)
<function prev_month at 0x810adf4>
(2001, 12)
<function <lambda> at 0x80db82c>
(2001, 12)
```

Другой вопрос касается того, как же именно значения передаются функции. Здесь действуют те же правила, что и для присваиваний. При передаче в функцию неизменяемых типов (чисел, строк, кортежей) создаются их копии, тогда как для более сложных, изменяемых типов (списков, словарей, файлов и других объектов) передаются их оригиналы. (В терминологии языка C в первом случае имеет место передача по значению, а во втором — по ссылке). Разумеется, если функция изменяет что-то в переданном ей объекте, то эти изменения отразятся и на исходном объекте, ведь формальный аргумент — просто другое имя для переданного фактического аргумента.

Листинг 8.7. Передача значений функции

```
def prev_month(y_m):
   """ Следующий месяц изменяем на месте, в списке """
   if y_m[1] == 1:
     y_m[0] = y_m[0] - 1
     y_m[1] = 12
   else:
     y_m[1] = y_m[1] - 1

def fake_inc(x):
   """ С целым числом такое не пройдет """
   x += 1

year_month = [2001,4]
print year_month
prev_month(year_month)
print year_month
year = 2001
```

```
print year
fake_inc(year)
print year
```

Результат выполнения:

```
[2001, 4]
[2001, 3]
2001
2001
```

Функции, как и любой другой тип данных, можно передавать в качестве аргументов и возвращать по оператору `return`. В терминах функционального программирования функции Python могут работать как *функции высшего порядка*, что приводит к очень компактной записи алгоритмов, особенно рекурсивных. Следующий пример демонстрирует более простой подход — передачу функции в качестве аргумента.

Листинг 8.8. Передача функции в качестве аргумента

```
def supermap(f, lst):
   """Упрощенный рекурсивный аналог map."""
   from operator import isSequenceType
   res = []
   for i in lst:
      if isSequenceType(i):
         res.append(supermap(f, i))
      else:
         res.append(f(i))
   return res


print supermap.__doc__
l = [1, -1, [2, [2, -3]], -8]
print l
print supermap(abs, l)
```

Результат выполнения:

```
Упрощенный рекурсивный аналог map
[1, -1, [2, [2, -3]], -8]
[1, 1, [2, [2, 3]], 8]
```

Функция как объект имеет несколько (не очень часто используемых) атрибутов. Они перечислены в табл. 8.7.

Таблица 8.7. *Атрибуты объекта-функции*

Название атрибута	Смысл атрибута
`__doc__`, `func_doc`	Строка документации
`__name__`, `func_name`	Название функции
`func_code`	Кодовый объект, содержащий код функции *(см. статью "Кодовые объекты" в разд. 8.2.10)*
`func_defaults`	Значения по умолчанию
`func_globals`	Глобальное пространство имен для данной функции

Объект-функция фактически является оберткой вокруг соответствующего исполняемого кода `func_code`, содержащей некоторую контекстную информацию.

Благодаря грамотно написанным строкам документации с функциями легко работать в диалоговом режиме.

Листинг 8.9. Использование строк документации в диалоговом режиме

```
>>> print id.__doc__
id(object) -> integer

Return the identity of an object.  This is guaranteed to be unique among
simultaneously existing objects.  (Hint: it's the object's memory ad-
dress.)
>>> id([1,2,3])
135343692
```

В Python 2.1 функциям можно присваивать произвольные атрибуты:

```
def f(x):
  pass

f.info = "empty function"
f.secure = 1
```

А в Python 2.2 появится специальный вид функции для порождения объектов-генераторов *(см. разд. 8.11)*.

Методы

Методы — функции, определяющие операции над объектами. От функций их отличает место описания, синтаксис вызова, трактовка первого аргумента и специфика применения. Метод описывается с помощью оператора `def`

непосредственно в блоке `class` и обратиться к нему можно как и к атрибуту объекта. Первый формальный аргумент метода (как бы он ни назывался) заменяется на объект, для которого метод вызван. Обычно первый аргумент имеет имя `self`, чтобы отразить этот факт. Поэтому фактических аргументов нужно задавать на один меньше, чем формальных. Определенный по `class` объект может участвовать и в операциях, подобных операциям над объектами встроенных типов. Для этого он должен иметь соответствующие специальные методы, имена которых окружены двойными подчеркиваниями "__". В отличие от функций, метод служит для задания операций над объектами конкретного класса (или классов) и ему разрешено изменять состояние объекта (или класса), для которого он определен. В языке Python не реализовано разделение методов на общедоступные, защищенные и внутренние, как в некоторых других объектно-ориентированных языках, поэтому ограничения доступа к методам объекта обеспечиваются на уровне стиля программирования. Обычно внутренние методы начинаются с одиночного подчеркивания.

Примечание

Защищенным методам можно дать имя, начинающееся двумя подчеркиваниями "__", но не заканчивающееся на "__" *(см. разд. 8.2.6)*.

Листинг 8.10. Описание класса и его методов

```
class A:
  """Класс с примерами методов"""
  def __init__(this, x):
    """Конструктор для данного класса"""
    this.main_value = x

  def delta(self, x):
    """Метод, изменяющий состояние объекта"""
    self.main_value += x
    return self.main_value

  def __radd__(self, other):
    """Специальный метод для сложения объекта с другим"""
    return A(self.main_value+other.main_value)

  def __str__(self):
    """Метод, представляющий объект в виде строки"""
    return "A: "+str(self.main_value)
```

```
a=A(1)
b=A(2)
a.delta(5)
print a.delta
l = [1, 2, 3]
print l.append
print a+b
```

Результат выполнения:

```
<method A.delta of A instance at 0x80db8dc>
<built-in method append of list object at 0x80db95c>
A: 8
```

Объект-метод является надстройкой над объектом-функцией. Атрибуты объекта-метода приведены в табл. 8.8.

Таблица 8.8. *Атрибуты объекта-метода*

Название атрибута	Смысл атрибута
`__doc__`, `func_doc`	Строка документации
`__name__`, `func_name`	Название метода
`im_func`	Объект-функция, соответствующая методу
`im_class`	Класс, в котором определен метод
`im_self`	Объект, которому принадлежит метод, или `None`

Фактически вызов метода объекта `obj.m(args,...)` идентичен вызову функции метода с указанием объекта в качестве первого аргумента `obj.m.im_func(obj, args)`.

8.2.6. Классы и объекты

Класс (class) тоже является объектом. Основное назначение класса — обеспечивать создание своих объектов и служить базовым классом для других классов. Атрибуты и методы класса доступны и как атрибуты объектов данного класса. В табл. 8.9 перечислены имеющиеся у класса атрибуты.

Таблица 8.9. *Встроенные атрибуты класса*

Название атрибута	Смысл атрибута
`__doc__`	Строка документации
`__module__`	Модуль, которому принадлежит класс
`__dict__`	Особый словарь, содержащий таблицу символов модуля

Листинг 8.11. Файл obj_attrs.py. Объект и его специальные атрибуты

```
class A:        # минимально возможный класс
    pass

class B(A):     # наследник минимально возможного класса
    """B derived from A."""
    pass

a = A()         # объект класса A
print "type:", map(type,[A, B, a])
print "repr:", `A`, `B`
b = B()
print "__doc__:", B.__doc__
print "__module__:", B.__module__
print "__dict__:", B.__dict__
```

Результат выполнения:

```
type: [<type 'class'>, <type 'class'>, <type 'instance'>]
repr: <class __main__.A at 0x80db82c> <class __main__.B at 0x80d279c>
__doc__: B derived from A.
__module__: __main__
__dict__: {'__module__': '__main__', '__doc__': 'B derived from A.'}
```

Объект (object) или *экземпляр класса (class instance)* получается в результате вызова соответствующего класса на исполнение. Специальный *метод-конструктор (constructor)* `__init__()` позволяет инициализировать значение объекта. *Метод-деструктор (destructor)*, вызываемый непосредственно перед удалением объекта, имеет название `__del__()` и служит для выполнения действий, требуемых для корректного освобождения ресурсов (окон, открытых файлов и т. п.).

Объект, определенный с помощью оператора `class`, всегда имеет атрибуты, указанные в табл. 8.10.

Таблица 8.10. *Атрибуты объекта, определенного оператором* `class`

Название атрибута	Смысл атрибута
`__doc__`	Строка документации (класса)
`__class__`	Класс, которому принадлежит объект
`__module__`	Модуль, которому принадлежит класс объекта

Таблица 8.10 *(окончание)*

Название атрибута	Смысл атрибута
`__dict__`	Особый словарь, содержащий таблицу символов объекта

Время между созданием и удалением объекта из памяти называется *временем жизни объекта*.

Листинг 8.12. Файл obj_life.py. Создание и удаление объектов

```
class AA:
    def __init__(self, x):
        self.x = x
        print "Создание", self.x
    def __del__(self):
        print "Удаление", self.x


a = AA(1)
b = a
c = AA(2)
print "Объект 2 теряет единственное имя c:"
c = 2+2
print "Объект 1 теряет связь с именем a (остается b):"
del a
print "Объект 1 теряет связь с именем b:"
del b
```

Результат выполнения:

```
Создание 1
Создание 2
Объект 2 теряет единственное имя c:
Удаление 2
Объект 1 теряет связь с именем a (остается b):
Объект 1 теряет связь с именем b:
Удаление 1
```

Следующий пример показывает как из-за циклических ссылок объекты могут не удаляться из памяти и что можно сделать во избежание этой ситуации:

Листинг 8.13. Файл cyclic_ref.py. Проблема с удалением объектов из-за циклических ссылок

```
class AA:
    def __init__(self, x, other=None):
        self.x = x
        # если other не задан, создаем его
        self.link = other or AA(x+1, self)
        print "Создание пары", self.x, self.link.x
    def __del__(self):
        print "Удаление", self.x


a = AA(1)
del a
b = AA(3)
print "Удалим 4:"
del b.link
print "Удалим 3:"
del b
```

Результат выполнения:

```
Создание пары 2 1
Создание пары 1 2
Создание пары 4 3
Создание пары 3 4
Удалим 4:
Удаление 4
Удалим 3:
Удаление 3
```

Здесь объекты 1 и 2 ссылаются друг на друга, и потому их не может уничтожить удаление имени, связанного с объектом 1. Пара объектов 3 и 4 удаляется из памяти после предварительного удаления ссылки на объект 4 из объекта 3, после чего объект 3 можно удалить, т. к. на него больше нет ссылок, кроме как по имени `b`.

Циклические ссылки очень опасны в программах, которые работают, не выгружаясь из памяти, создавая *"утечки памяти" (см. также разд. 8.2)*. Это ведет к неконтролируемому росту необходимой для приложения памяти и мо-

жет привести его к краху. Циклические ссылки — плохое решение с точки зрения объектного дизайна. Например, в приведенной выше ситуации можно создать специальный класс-контейнер, который бы содержал пару объектов и управлял их существованием. Такой подход приводит не только к исключению циклических ссылок, но и к упрощению кода и более широким возможностям его повторного использования.

Листинг 8.14. Файл cyclic_ref1.py. Решение проблемы с удалением объектов

```
class AA:
    def __init__(self, x, other=None):
        self.x = x
        print "Создание", self.x
    def __del__(self):
        print "Удаление", self.x

class CAA:
    def __init__(self, one, other):
        self.pair = (AA(one), AA(other))
        print "Создание пары", one, other
    def __del__(self):
        print "Удаление пары", self.pair[0].x, self.pair[1].x

p = CAA(1,2)
del p
```

Результат выполнения:

```
Создание 1
Создание 2
Создание пары 1 2
Удаление пары 1 2
Удаление 2
Удаление 1
```

В объектно-ориентированном программировании принято скрывать некоторые атрибуты от пользователя. Как правило, это те атрибуты, которые хранят внутреннее состояние объекта или являются вспомогательными внутренними методами. В языке Python атрибуты класса или объекта недоступны извне под своим собственным именем, если это имя начинается двумя подчеркиваниями и не заканчивается двумя подчеркиваниями. Прав-

да, доступ к данному атрибуту все же возможен, но для этого к имени нужно добавить подчеркивание и имя класса в начале имени:

```
class Secretive:
  def __init__(self, mysecret):
    self.__mysecret = mysecret
  def give_secret(self):
    return self.__mysecret

s = Secretive(5)
print s.give_secret()  # Цивилизованный метод.
# print s.__mysecret   # А это вызовет ошибку.
print s._Secretive__mysecret  # Все можно обойти!
```

Однако такой способ применяется не очень часто. Как правило, достаточно одиночного подчеркивания в начале имени, чтобы программисты знали, что данный атрибут внутренний, и его не следует использовать за пределами методов класса.

8.2.7. Модули

Для программы на Python модуль — тоже объект, возникающий при выполнении оператора `import`. Он обеспечивает доступ к своему пространству имен с помощью обычной операции по доступу к атрибутам для чтения или записи. Объект-модуль имеет атрибут `__dict__`, который содержит *таблицу символов модуля* (его *пространство имен*). Словарю `__dict__` нельзя ничего присваивать, можно лишь изменять его отдельные элементы. Атрибуты объекта-модуля приведены в табл. 8.11.

Таблица 8.11. *Атрибуты объекта-модуля*

Название атрибута	Смысл атрибута
`__doc__`	Строка документации модуля
`__file__`	Имя файла с описанием модуля (`.pyc` или `.py`, в зависимости от того, существовал ли скомпилированный вариант при загрузке модуля)
`__name__`	Имя модуля
`__dict__`	Словарь, содержащий таблицу символов объекта
`__path__`	Путь к каталогу с пакетом (только если модуль — пакет)

Встроенная функция `dir()`, которой модуль задан в качестве аргумента, выдает список имен модуля.

```
import copy, sys

print copy
print sys
print "dir(copy):", dir(copy)
print "version:", sys.version
```

Результат выполнения:

```
<module 'copy' from '/usr/local/lib/python2.0/copy.pyc'>
<module 'sys' (built-in)>
['Error', '_EmptyClass', '__builtins__', '__doc__', '__file__',
'__name__', '_copy_atomic', '_copy_dict', '_copy_dispatch', '_copy_inst',
'_copy_list', '_copy_tuple', '_deepcopy_atomic', '_deepcopy_dict',
'_deepcopy_dispatch', '_deepcopy_inst', '_deepcopy_list',
'_deepcopy_tuple', '_keep_alive', '_test', 'copy', 'deepcopy', 'error']
2.0 (#1, Oct 16 2000, 18:10:03)
[GCC 2.95.2 19991024 (release)]
```

(См. разд. 8.8.)

8.2.8. Тип

Для представления типа в Python имеется свой тип данных — *объект-тип*. Порождается встроенной функцией `type()`, которая возвращает тип данного ей аргумента. Объект этого типа не имеет никаких методов. Объект-тип имеет атрибут `__name__`, который дает строку с названием типа.

```
all = [1, 2L, 3.0, 4j, "5", (6,), [7], {8:"dict"}]
print "type(1):", type(1)
print map(type, all)
print "type(type(1)).__name__:", type(type(1)).__name__
```

Результат выполнения:

```
type(1): <type 'int'>
[<type 'int'>, <type 'long int'>, <type 'float'>, <type 'complex'>, <type
'string'>, <type 'tuple'>, <type 'list'>, <type 'dictionary'>]
type(type(1)).__name__: type
```

Модуль `types` дает объекты-типы для различных встроенных типов.

```
>>> from types import *
>>> if type([1, 2]) == ListType:
```

```
...     print "список"
...
список
```

До Python 2.2 в языке существовало различие между типами и классами. Например, строки не имели своего класса. Начиная с Python 2.2, указанное отличие начинает стираться: встроенные типы получают классы, от которых можно наследовать, как и от "классических" классов.

8.2.9. Файлы

Файловые объекты используются для ввода и вывода в файлы и сохранения связанной с файлами контекстной информации. Атрибуты этого объекта примерно соответствуют стандартному stdio из языка C. Объект порождается встроенной функцией `open()`, а также некоторыми другими функциями и методами (`os.popen()`, `os.fdopen()` и т. п.). Помимо собственно файловых объектов используются и *файлоподобные объекты*, реализующие не все методы, а только интересные с практической точки зрения.

Объект-файл имеет следующие атрибуты.

- `close()`

  Закрывает файл. После закрытия нельзя читать из файла или писать в файл. На операциях, требующих чтобы файл был открыт, возбуждается исключение `IOError`. Этот метод автоматически вызывается при удалении файлового объекта из памяти. См. атрибут `closed` ниже.

- `closed`

  Свойство характеризует текущее состояние файла. Равно `0` для открытого файла и `1` для закрытого. Атрибуту нельзя присвоить значение.

- `flush()`

  Очистка внутреннего буфера, связанного с файлом. Все еще не записанные находящиеся в буфере данные записываются.

- `isatty() -> boolean`

  Возвращает истинное значение `boolean`, если файл является терминальным устройством (линией). Например, файл стандартного вывода `sys.stdout.isatty()` обычно дает `1` (если вывод не перенаправлен, а идет сразу на терминал).

- `fileno() -> int`

  Возвращает целочисленный номер файла, использующийся операционной системой для низкоуровневых операций ввода/вывода.

- `mode`

  Режим ввода/вывода для данного файла. Для файлов, открытых по функции `open()` — аргумент `mode`. Только для чтения.

- `name`

  Для обычных файлов — имя файла. Для других файлов указывает происхождение файлового объекта в угловых скобках. Например, `sys.stdio.name == '<stdio>'`. Только для чтения.

- `read([size]) -> string`

  Читать из файла не более `size` байтов (сохраняются в строке `string`). Если `size` не задан или меньше нуля — читать файл до конца. Возвращение пустой строки означает конец файла. Для некоторых видов файлов (вроде терминальных устройств) можно пробовать читать и после конца файла.

- `readline([size]) -> string`

  Данный метод читает строку из файла. Конец строки определяется символом новой строки, концом файла или максимальным заданным размером. В конце строки стоит `"\n"`, если он влезает в длину `size`. Символ `"\n"` может отсутствовать и у последней строки файла. Как и `read()`, метод возвращает пустую строку при достижении конца файла. В отличие от языка C, в строке могут присутствовать символы с кодом `"\0"`.

- `readlines([approx_size]) -> list`

  С помощью этого метода можно прочитать все строки файла в список `list`. Аргумент `approx_size` позволяет задать желательный приблизительный общий размер всех читаемых строк.

- `seek(offset[, mode])`

  Смещает текущую позицию в файле на значение `offset` относительно начала файла (`mode` равен `0`, действует по умолчанию параметра `mode`), относительно текущей позиции (`mode == 1`), относительно конца файла (`mode == 2`). Для последовательных файлов (вроде терминальных устройств, tty) эту функцию применять нельзя, она возбуждает исключение `IOError`.

- `softspace`

  Индикатор необходимости выводить пробел перед следующим печатаемым значением. Должен быть доступен для записи.

- `tell() -> int`

  Говорит о текущей позиции в файле. Начало файла соответствует нулю.

- `truncate([size])`

  Обрубает файл до текущей позиции или до указанного размера `size`. Действует не на всех платформах.

- `write(string)`

  Метод служит для буквальной записи строки в файл. В отличие от `print`, нужно самостоятельно заботиться о разделителях строк (`"\n"`). В зависи-

мости от настроек буферизации строка может попасть в файл не сразу, а только после вызова метода `flush()` или закрытия файла.

- `writelines(list)`

  Записывает строки из списка в файл. Как и во `write()`, разделители строк не пишутся. Метод `writelines()` совместим с `readlines()`, поэтому `f2.writelines(f1.readlines())` должен переписать файл `f1` в файл `f2` один к одному.

Файлоподобные объекты зачастую имеют только методы `close()`, `read()`, `write()`, `readlines()`, `writelines()` (в зависимости от назначения). Остальные атрибуты могут отсутствовать. В любом случае лучше заранее продумать, будет ли код поддерживать "нестандартные" файлы или нет. Начиная с Python 2.0, встроенный оператор `print` умеет работать с файлоподобными объектами *(см. статью "Оператор print" в разд. 8.5.1)*.

```
import sys
f = open("somefile.tmp", "wt+")
print "Свойства sys.stdin, sys.stdout, f:"
print "mode:", sys.stdin.mode, sys.stdout.mode, f.mode
print "name:", sys.stdin.name, sys.stdout.name, f.name
print "isatty():", sys.stdin.isatty(), sys.stdout.isatty(), f.isatty()
print "fileno():", sys.stdin.fileno(), sys.stdout.fileno(), f.fileno()
print "closed:", sys.stdin.closed, sys.stdout.closed, f.closed
```

Результат выполнения:

```
Свойства sys.stdin, sys.stdout, f:
mode: r w wt+
name: <stdin> <stdout> somefile.tmp
isatty(): 1 1 0
fileno(): 0 1 3
closed: 0 0 0
```

Значение `isatty()` для стандартного ввода/вывода зависит от того, как исполняется приведенный код. При перенаправлении ввода или вывода в файл stdin или stdout перестанут быть tty-устройствами и `isatty()` примет значение 0.

```
Суммирование чисел из небольшого файла
"""Суммируем числа из не очень большого файла.
По одному числу на строку.
"""
# Для не очень большого файла
numbers = open("numbers.dat", "rt")
```

```
S = 0
for n in numbers.readlines():
  try:
    S = S + float(n)
  except:
    pass # не число молча пропускаем
print S
```

Другой пример делает то же самое, но с меньшими затратами оперативной памяти.

```
Суммируем числа из большого файла
"""Суммируем числа из (возможно) очень большого файла.
По одному числу на строку.
"""
# Для не очень большого файла
numbers = open("numbers.dat", "rt")
S = 0
while 1:
  n = numbers.readline()
  if not n:
    break
  try:
    S = S + float(n)
  except:
    pass # не число молча пропускаем
print S
```

8.2.10. Другие встроенные типы

Помимо перечисленных выше, Python и его расширения используют другие объекты, которые встречаются в явном виде в программах не так часто, но стоят того, чтобы о них упомянуть. Python предоставляет достаточно широкие возможности для работы со своим внутренним устройством. Перечисленные далее объекты используются, в основном, самими разработчиками Python и его модулей, но будут интересны и тем, кому по той или иной причине необходимы столь низкоуровневые интерфейсы и внутренности языка Python. Данная книга не дает полной информации в этом направлении, поэтому интересующимся будет полезно посмотреть исходные тексты соответствующих модулей.

Предупреждение

Реализация описанных далее встроенных объектов, их атрибуты и механизмы взаимодействия могут измениться в будущих реализациях Python. В случае необходимости их использования следует тщательно локализовать места их использования в отдельном модуле (или модулях), чтобы переход на другую версию Python был как можно более безболезненным.

Объекты-срезы

При использовании нотации выделения среза *(см. статью "Индексные выражения" в разд. 8.3.2)* интерпретатор генерирует *объект-срез* (или кортеж таких объектов), который он передает методу `__getslice__()`, `__setslice__()` или `__delslice__()`. Объект-срез содержит три атрибута: `start` — начало, `stop` — конец (как обычно в Python — исключая элемент с указанным в конце номером) и `step` — шаг, с которым берутся элементы среза из исходного объекта. Создать объект-срез можно с помощью функции `slice()` *(см. статью "Другие функции" в разд. 8.6.1)*. Срезы очень важны для численных и научных приложений Python, например, *см. гл. 11*. Следующий пример дает хорошее представление о том, как их применять:

Листинг 8.15. Примеры срезов

```
class SliceExample:
  def __len__(self):
    return 100
  def __getitem__(self, x):
    try:
      details = ": start=%s, stop=%s, step=%s" % (x.start, x.stop,
x.step)
    except:
      details = ""
    return str(x) + details

s = SliceExample()
examples = ["s[1]", "s[1:10]", "s[:]", "s[:-3]",
           "s[::2]", "s[50::2]", "s[1, 2, 23]",
           "s[1, 1:2, 1:90:5]",
           "s['a':'z']"      # ошибка
            ]
for e in examples:
  print "%11s: " % e,
```

```
  try:
    print "%s" % (eval(e),)
  except TypeError, x:
    print "TypeError: %s" % x
```

Результат выполнения:

```
       s[1]:  1
     s[1:10]:  slice(1, 10, None): start=1, stop=10, step=None
       s[:]:  slice(0, 2147483647, None): start=0, stop=2147483647,
step=None
     s[:-3]:  slice(0, 97, None): start=0, stop=97, step=None
     s[::2]:  slice(None, None, 2): start=None, stop=None, step=2
   s[50::2]:  slice(50, None, 2): start=50, stop=None, step=2
s[1, 2, 23]:  (1, 2, 23)
s[1, 1:2, 1:90:5]:  (1, slice(1, 2, None), slice(1, 90, 5))
 s['a':'z']:  TypeError: slice index must be int
```

Кодовые объекты

Как уже говорилось, интерпретатор Python сначала компилирует блоки кода в промежуточный байт-код, а затем уже интерпретирует его. *Кодовые объекты* хранят скомпилированный код. Кодовый объект функции можно получить либо как атрибут `func_code` уже определенной функции *(см. статью "Функции" в разд. 8.2.5)*, либо скомпилировать из исходного текста с помощью функции `compile()` *(см. статью "Функции компиляции и исполнения" в разд. 8.6.1)*. Кодовые объекты сами неизменяемые и содержат ссылки на изменяемые объекты. В табл. 8.12 приведены некоторые атрибуты кодового объекта.

Таблица 8.12. *Некоторые атрибуты кодового объекта*

Название атрибута	Смысл атрибута
`Co_argcount`	Число аргументов
`Co_code`	Строка, представляющая байт-код
`Co_consts`	Константы, использованные в блоке кода
`Co_filename`	Источник кода
`Co_firstlineno`	Номер первой строки кода в файле
`Co_name`	Имя блока кода (например, имя функции)
`Co_names`	Имена, использованные в байт-коде
`Co_nlocals`	Число используемых локальных переменных (включая аргументы)
`Co_varnames`	Имена локальных переменных (включая аргументы)

Примечание

Полный список атрибутов кодового объекта можно получить с помощью функции `dir()`, примененной к кодовому объекту.

Фрейм-объекты

Фрейм-объекты предоставляют доступ к данным фреймов исполнения *(см. разд. 8.7)* и используются в трассировочных объектах. Фрейм-объекты хранят контекст исполнения кода. Их можно использовать в обработчиках исключений, отладчиках, профайлерах. Атрибуты фрейм-объектов указаны в табл. 8.13.

Таблица 8.13. *Некоторые атрибуты фрейм-объектов*

Название атрибута	Смысл атрибута
`f_back`	Предыдущий фрейм
`f_builtins`	Встроенные имена
`f_globals`	Глобальные имена
`f_locals`	Локальные имена
`f_lineno`, `f_lasti`	Номер строки и инструкции
`f_code`	Кодовый объект, исполняемый в данном фрейме
`f_trace`	Функция, которая будет вызываться перед каждой строкой исходного кода (по умолчанию `None`)
`f_exc_type`, `f_exc_value`, `f_exc_traceback`	Данные о последнем исключении, произошедшем в данном фрейме

Пример работы с фреймами исполнения можно посмотреть в исходных текстах модулей `pdb`, `bdb`, `traceback`.

Трассировочные объекты

Трассировочные объекты представляют содержимое стека исполнения от места возбуждения исключения до места его обработки. В обработчике исключений соответствующий исключению трассировочный объект доступен как `sys.exc_info()[2]`. В табл. 8.14 приведены атрибуты трассировочного объекта.

Таблица 8.14. *Некоторые атрибуты трассировочного объекта*

Название атрибута	Смысл атрибута
`tb_frame`	Фрейм исполнения текущего уровня

Таблица 8.14 *(окончание)*

Название атрибута	Смысл атрибута
`tb_lineno` и `tb_lasti`	Номер строки и инструкции, где случилось исключение
`tb_next`	Следующий уровень стека (другой трассировочный объект)

Для работы с трассировочным объектом наиболее удобно воспользоваться поставляемым вместе с Python модулем `traceback` *(см. разд. 9.2.11)*.

8.3. Выражения

Особое место среди конструкций любого языка программирования высокого уровня занимают *выражения (expressions)*. Несложные выражения в Python мало чем отличаются от выражений на Pascal, C, Java или Basic по внешнему виду и не обманут интуиции программистов. Обозначения операций тоже, в основном, удовлетворяют неписанному стандарту. Конечно, имеются и различия, и свойственные только Python правила применения операций.

Например, в Python нет отдельного типа для логических величин. Логической ложью считаются нули, пустые строки, списки, кортежи, словари, объект `None`, и заданные пользователем объекты с определенными свойствами *(см. разд. 8.4)*. Все остальное — логическая истина. Однако "правильными" логическими значениями, получающимися в результате отрицания, сравнений и других подобных проверок являются `0` и `1`. Таким образом, `not not [1,2,3]` станет просто `1`, а вот `[1,2,3] or [4,5,6]` будет равно `[1,2,3]` *(см. статью "Логические операции" в разд. 8.3.3)*.

Еще одним полезным исключением из правил являются операции сравнения, которые группируются в цепочки:

```
>>> if 1 < 2 < 3 <= 4: print 1
1
```

Примечание

Здесь и далее мы будем видеть выражения в составе других операторов, в основном, `print`. Не стоит забывать, что выражение может быть и самостоятельным оператором.

Выражения состоят из *операций* и *операндов*. Операции в Python бывают *унарными* (требующими одного операнда) и *бинарными* (требуют двух операндов).

Например:

```
a = 1; b = 2; c = [1,2,3]
print not a
print a and b
print c[a:b]
```

Результат выполнения:

```
0
2
[2]
```

В третьем выражении мы имеем дело не с одной операцией, а с двумя. Сначала выполняется порождение объекта-среза `slice(1,2,None)`, а затем операция выделения среза из списка `c`. Но эти операции в синтаксисе Python так, что выглядят как одна операция с тремя аргументами (тернарная операция).

Для бинарных операций встает вопрос об их *ассоциативности*. Почти все бинарные операции ассоциативны слева направо (что не противоречит интуиции). Для коммутативных операций (то есть операций, результат которых не изменяется, если поменять операнды местами), ассоциативность во многих случаях не играет роли.

Замечание

Правда, даже от перемены мест слагаемых сумма, например, чисел с плавающей точкой может измениться. Это связано с тем, что вычисления производятся с конечной точностью. В таких ситуациях для достижения большей точности следует сначала складывать меньшие числа и лишь затем числа побольше.

Еще один пример:

```
>>> a = 2000000000
>>> b = 200000000
>>> c = -100000000
>>> a + b + c
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
OverflowError: integer addition
>>> a + c + b
2100000000
```

На этом примере мы наглядно видим, что от порядка сложения может зависеть исход операции.

Ассоциативность слева-направо означает, что в ряду одинаковых операций действия будут выполняться слева направо:

```
a+b+c+d == ((a+b)+c)+d
```

Вычитание тоже ассоциативно слева направо. Сравните:

```
>>> print 1-2-3-4, ((1-2)-3)-4, 1-(2-(3-4))
-8 -8 -2
```

Исключением, ассоциирующимся наоборот, справа налево является операция возведения в степень:

```
>>> print 2L**3L**4L, (2L**3L)**4L, 2L**(3L**4L)
2417851639229258349412352 4096 2417851639229258349412352
```

Такой порядок используется в математике при вычислении цепочек возведений в степень. (Заметьте, что для избавления от переполнений пришлось использовать длинные целые, о чем говорит `L`, поставленная после чисел).

Еще одним важным вопросом является приоритет операций. Любой человек, хоть немного знакомый с арифметикой, без труда скажет, что `2+3*4 == 14`, а не `20`. Это связано с принятыми в арифметике приоритетами операций. Умножение имеет бóльший приоритет, чем сложение. Однако скобка имеет еще бóльший приоритет, и потому `(2+3)*4 == 20`. С помощью скобок можно явно обойти принятые приоритеты и определить порядок выполнения операций.

Замечание

Когда ассоциативность или приоритет операций вам неизвестны, лучше задать порядок вычислений с помощью скобок.

Принятые в Python приоритеты операций даны в табл. 8.15.

Таблица 8.15. *Приоритеты операций*

Операция	Описание
`lambda`	Лямбда-выражение
`Or`	Логическое ИЛИ
`And`	Логическое И
`Not x`	Логическое НЕ
`in, not in`	Проверка принадлежности
`is, is not`	Проверка идентичности
`<, <=, >, >=, <>, !=, ==`	Сравнения
`|`	Побитовое ИЛИ

Таблица 8.15 (окончание)

Операция	Описание
^	Побитовое исключающее ИЛИ
&	Побитовое И
<<,>>	Сдвиги
+, -	Сложение и вычитание
*, /, %	Умножение, деление, остаток
+x, -x	Унарный плюс и смена знака
~x	Побитовое НЕ
**	Возведение в степень
x.атрибут	Ссылка на атрибут
x[индекс]	Взятие элемента по индексу
x[от:до]	Выделение среза (от и до)
f(аргумент,...)	Вызов функции
(...)	Скобки или кортеж
[...]	Список или списковое включение
{ключ:данное,...}	Словарь
`выражения,...`	Преобразование в строку

Остается только отметить, что операции с одинаковым приоритетом подчиняются правилам ассоциации. В связи с этим стоит рассмотреть поучительный пример:

```
>>> print "%s" % 5*2
55
```

Дело в том, что несмотря на различные типы данных, к которым применяются операции, их правила ассоциативности остаются в силе. Таким образом, для получения ожидаемого результата надо было написать:

```
>>> print "%s" % (5*2)
10
```

Порядок вычислений операндов в Python можно определить следующими правилами:

- В любых бинарных операциях (кроме возведения в степень) операнд, стоящий слева, выполняется раньше операнда, стоящего справа. Серия возведений в степень выполняется справа налево.

- Цепочка сравнений вида `a op1 b op2 c ... y opn z` фактически является сокращением для: `(a op1 b) and (b op2 c) and ... and (y opn z)`.
- Перед фактическим выполнением операции вычисляются нужные для нее операнды. Для большинства бинарных операций вычисляются оба операнда (сначала левый), однако для операций `or`, `and` и в цепочке сравнений иногда достаточно вычисления только одного операнда. В этом случае второй операнд вообще не будет вычисляться. (В нем даже могут быть неопределенные переменные, что вызвало бы ошибку, будь операнд вычислен).
- Аргументы функций и выражения для списков и т. п. вычисляются слева направо, т. е. в порядке следования в выражении.

Как всегда, примеры:

```
>>> global v
>>> v = 0
>>> def t():       # для начала определим функцию с побочным эффектом
...     global v
...     v = v + 1    # позволяющим выявить порядок вычислений
...     return v
...
>>> def f(x,y,z):   # пробная функция
...     print x,y,z
...
>>> print t() > t() > t() > t(), v   # первого сравнения хватило для
0 2                                  # вычисления результата
>>> v=0; print t() ** t() ** t()     # сначала вычисляется самый левый
операнд
1                                    # == 1 ** 2 ** 3
>>> v=0; print t() + t() * t()       # здесь тоже:
7                                    # == 1 + 2 * 3
>>> v=0; print t() or t(), v         # здесь второй t() даже не вызывался
1 1
>>> v=0; print t() or ttt()          # более того, до явной ошибки
1                                    # (ttt -- неизвестная функция!)
                                     # дело не доходит
>>> v=0; f(z=t(),y=t(),x=t())        # все равно t() вычисляются
3 2 1                                # слева-направо
# Здесь функция вызывается с явным указанием имен формальных параметров.
>>> v=0; print {t():t(),t():t()}     # сначала вычисляется значение,
```

```
{2: 1, 4: 3}                          # затем -- ключ
>>> v=0; print "12345"[t():t()]       # здесь все в порядке
2                                     # == "12345"[1:2]
>>> v=0; print [t(),t(),t(),t()]      # и здесь все в порядке
[1, 2, 3, 4]
```

На практике использование функций с побочными эффектами не поощряется. Это снижает читаемость кода и может вызвать очень коварные логические ошибки.

8.3.1. Атомы

Атомами (atoms) условно называют наиболее "плотные" синтаксические элементы выражений. К атомам относятся имена переменных, литералы простейших встроенных типов и элементы, заключенные в круглые, квадратные или фигурные скобки, а также в обратные апострофы. Эти конструкции имеют наибольший приоритет при вычислениях. В данном разделе мы рассмотрим их по порядку.

Идентификаторы

Идентификатор (identifier) или просто *имя* — строка символов, используемая для идентификации элемента программы (объекта, имени функции и т. п.). В Python идентификаторы имеют следующий синтаксис:

```
идентификатор:      (буква|"_") (буква|цифра|"_")*
буква:              "a"..."z"|"A"..."Z"
цифра:              "0"..."9"
```

То есть они должны начинаться с буквы или подчеркивания, за которыми могут следовать буквы, цифры и подчеркивания. Регистр букв имеет значение: `apple` и `Apple` — разные имена. Длина имен в Python неограничена. Некоторые имена нельзя использовать, т. к. они являются зарезервированными. Вот они:

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass
def       finally   in        print
```

Получить их можно так:

```
>>> import keyword
>>> keyword.kwlist
```

```
['and', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or',
'pass', 'print', 'raise', 'return', 'try', 'while']
```

В табл. 8.16 приведены классы имен, имеющих специальное значение в Python (вместо "*" — буквы, цифры; внутри могут быть и подчеркивания).

Таблица 8.16. *Специальные классы имен в Python*

Шаблон имени	Специфика имени	Примечание
_	При интерактивном использовании Python содержит результат предыдущего вычисления	Оно же по традиции используется для перевода строк при разработке интернациональных программ *(см. разд. 9.21.2)*
_*	Не импортируется из модуля по команде "from module import *"	Подобные имена имеют вспомогательные, внутренние объекты в модулях и классах
__*__	Системное имя или имя для задания метода эмуляции встроенной операции	Например, метод __add__() позволяет эмулировать операцию "+"
__*	Используется для эмуляции защищенных (protected) атрибутов в классах	Например, метод __internal() класса A нельзя вызвать обращением A.__internal, т. к. вне методов данного класса оно превращается в A._A__internal

Программист сам выбирает имена для используемых им классов, переменных, функций и т. д. Существуют различные стили написания кода *(см. разд. 8.10)*. Некоторые из них требуют писать что-то вроде переменной, например, iMaxLength для обозначения целой "Максимальная длина", другие настоятельно рекомендуют max_length. В любом случае, для имен, используемых более, чем в 15 идущих подряд строках кода, рекомендуется давать осмысленные имена. (Современные редакторы типа Emacs умеют довводить вводимые слова по первым буквам так, что длинные имена при использовании таких редакторов не замедлят процесс написания кода.) В некоторых случаях короткие имена типа i или S, наоборот, полезны, т. к. говорят программисту о локальности своего существования внутри данного цикла или определения функции.

Также нежелательно, чтобы имена конфликтовали с именами встроенных функций. Например, часто забывают о существовании встроенной функции max и дают подобное имя переменной, тем самым делая невозможным ее

использование по имени max. (Интерпретатор при этом не ругается, ведь max не является зарезервированным именем).

Константы (литералы)

Константы служат для введения в программу фиксированных значений некоторых простых встроенных типов. В языке Python практически нет именованных констант, в связи с чем по-настоящему константами являются только *литералы* — величины, значение которых определяется их обозначением в программе. Слово "литерал" будет использоваться только для подчеркивания синтаксического происхождения константы.

Примечание

В этой книге слово "константа" будет применяться и по отношению к именам, значение которых не имеет смысла изменять. Например, константа math.pi на самом деле является переменной, но используется только как константа.

В Python отдельные литералы существуют для различных строк, положительных целых и длинных целых чисел, положительных чисел с плавающей точкой и мнимых чисел. Например, не существует отдельного литерала для отрицательных чисел — отрицательные числа вводятся как результат операции смены знака для целых констант. Этот факт не должен смущать: смысл записи выражения от этого не меняется.

Строковые константы. Для введения в программу *строковых констант* можно использовать апострофы, кавычки, утроенные апострофы или кавычки. Разумеется, начало и конец строки отмечается одинаковым типом кавычек. Назовем окруженные одиночными кавычками (апострофами) наборы символов *короткими строковыми литералами*, а утроенными кавычками (апострофами) — *длинными строковыми литералами*. Длинные строковые литералы могут содержать в себе кавычки, апострофы и даже переводы строк (как буквальные, так и заданные с помощью "\n"). В остальном они полностью аналогичны коротким строковым литералам.

Строки состоят из символов, входящих в таблицу *ASCII*, и символов с кодами 128—255, используемыми обычно для представления национальных алфавитов. В строковых литералах могут встречаться *специальные символы* (их иногда называют *управляющими*). В табл. 8.17 приведен список специальных символов.

Таблица 8.17. *Специальные символы*

Запись	Шестнадцатеричная запись	Значение
\\	\x5c	Обратная косая черта "\"
\'	\x2c	Апостроф "'"

Таблица 8.17 *(окончание)*

Запись	Шестнадцатеричная запись	Значение
`\"`	`\x22`	Двойная кавычка (")
`\a`	`\x07`	Звонок (BEL)
`\b`	`\x08`	Забой (BS)
`\f`	`\x0c`	Новая страница (FF)
`\n`	`\x0a`	Новая строка (LF)
`\r`	`\x0d`	Возврат каретки (CR)
`\t`	`\x09`	Горизонтальная табуляция (TAB)
`\v`	`\x0b`	Вертикальная табуляция (VT)
`\ddd`		ASCII-символ с восьмеричным значением `ddd` (не более трех цифр)
`\xdd...`		ASCII-символ с шестнадцатеричным значением `dd...` (любое количество цифр)
`\udddd`		Unicode-символ с шестнадцатеричным кодом `dddd` (только для Unicode-строк)
`\`новая строка		Игнорируется

Остальные сочетания обратной косой черты и некоторого символа передаются в строку "как есть".

Есть и еще одна разновидность строковых литералов — *необрабатываемые (raw) строковые литералы*. В них, в отличие от обычных строковых литералов, не действуют правила для введения специальных символов. Необрабатываемые строковые литералы обозначаются с помощью букв `r` или `R` перед открывающей кавычкой. Единственное ограничение на них состоит в том, что они не должны оканчиваться на одиночную обратную косую черту. Необрабатываемые строковые литералы очень удобны для задания регулярных выражений *(см. разд. 9.3.2)*.

И, наконец, последнее завоевание Python — строки Unicode. Соответствующие *строковые литералы Unicode* обозначаются префиксом `u` или `U`.

Здесь следует также отметить, что строковая константа, записанная на отдельной строке (или строках, если это длинная строковая константа) имеет самостоятельное значение — это строка документации. Следующие примеры покажут строковые константы, записанные различными видами строковых литералов. Заметьте, что литералы, записанные подряд, являются по сути

одним литералом, т. к. склеиваются еще при компиляции программы в байт-код.

```
def f1(x):
  "Строковая константа в кавычках (в ней действуют \\)"
  "Только первая из этих строк -- строка документации"
def f3(x):
  """Строковая константа
в утроенных кавычках"""
def fr(x):
  r"Сырая строковая константа (в ней не действуют \\)"
def fu(x):
  u'\u043A\u043E\u043C\u043C\u0435\u043D\u0442\u0430\u0440\u0438\u0439'

print f1.__doc__
print f3.__doc__
print fr.__doc__

import codecs
(encode,decode,sr,sw) = codecs.lookup('koi8-r')
print encode(fu.__doc__)[0]

print '''Теперь''' '''немного''' '''обычных''' '''констант'''
print 'Э' "т" '''и''' """ """ r"стр" "оки" r'''- одна строка'''
```

Результат выполнения:

```
Строковая константа в кавычках (в ней действуют \)
Строковая константа
в утроенных кавычках
Сырая строковая константа (в ней не действуют \\)
комментарий
Теперьнемногообычныхконстант
Эти строки- одна строка
```

При записи списков, состоящих из строковых констант, будьте внимательны с запятыми. Следующий пример показывает, что забытая запятая приводит не к ошибке, а к склеиванию соседних строк:

```
>>> l = ["a"
...   "b"]
>>> m = ["a",
...   "b"]
```

```
>>> l
['ab']
>>> m
['a', 'b']
```

Числовые константы. Запись целых чисел и чисел с плавающей точкой почти ни чем не отличается от принятой в других языках программирования нотации и очень похожа на используемую в языке C.

Целое число может быть записано не только в десятичной системе счисления, но и в восьмеричной и шестнадцатеричной. Для тех, кто использует только десятичную систему счисления, будет полезным напомнить, что ненулевое целое число нельзя начинать с цифры 0, т. к. это признак восьмеричного числа. Ниже приведен синтаксис записи целого числа, взятый из документации:

```
целое:              десятичное | восьмеричное | шестнадцатеричное
десятичное:         ("1"|...|"9") ("0"|...|"9")* | "0"
восьмеричное:       "0" ("0"|...|"7")+
шестнадцатеричное: "0" ("x"|"X")
("0"|...|"9"|"a"|...|"f"|"A"|...|"F")+
```

Запись длинного целого отличается от записи целого числа добавлением `"L"` (или `"l"`).

```
>>> print 0, 123, 001, 011, 0x10, 0xFF, 0Xff, 19999999999999999999L
0 123 1 9 16 255 255 19999999999999999999
```

Примеры неправильной записи (восьмеричный литерал не может иметь 9 в своей записи):

```
>>> print 009
  File "<stdin>", line 1
    print 009
            ^
SyntaxError: invalid syntax
>>>
```

Отрицательное число получается в результате операции смены знака, обозначаемой унарным минусом.

Запись числа с плавающей точкой имеет следующий синтаксис:

```
ЧПТ:         ЧПТ1 | ЧПТ2
ЧПТ1:        [десятичное] "." ("0"|...|"9")+ | десятичное "."
ЧПТ2:        (("1"|...|"9") ("0"|...|"9")* | ЧПТ1)
                                ("e"|"E") ["+"|"-"] ("0"|...|"9")+
десятичное: ("1"|...|"9") ("0"|...|"9")* | "0"
```

Следующие примеры помогут разобраться с синтаксисом:

```
>>> print 12.3, .123, 123., 0.0003, 1.23e-05, 0.123e5
12.3 0.123 123.0 0.0003 1.23e-05 12300.0
```

Примеры неправильной записи:

```
000.1e5 0,123
```

Комплексные константы получаются в результате суммирования числа с плавающей точкой и мнимого числа. Запись мнимого числа получается из записи числа с плавающей точкой (или целого десятичного числа) добавлением `"j"` или `"J"` в конце.

```
>>> print 10j, 123J, -1-3.2j, .1+.2j
10j 123j (-1-3.2j) (0.1+0.2j)
```

Задание кортежей, списков, словарей

Сложные встроенные типы данных — кортежи, списки, словари — имеют в Python хорошую синтаксическую поддержку.

Синтаксически *кортеж (см. статью "Кортежи" в разд. 8.2.3)* задается путем перечисления элементов через запятую внутри скобок:

```
кортеж:   "(" [элемент ","] ")" | "(" элемент ("," элемент)+ [","]  ")"
```

Элементом может быть любое Python-выражение. Пустой кортеж можно задать с помощью `"()"`. Одноэлементный кортеж нельзя просто написать в скобках, т. к. это означало бы просто элемент выражения, нужно обязательно писать запятую. Внутри скобок можно разрывать строку, например, указывая каждый элемент на отдельной строке. Рекомендуется ставить после каждого элемента запятую, т. к. в этом случае легко добавлять новые элементы, менять порядок существующих и удалять элементы, не заботясь о расстановке запятых. Элементы важных кортежей-констант, например, задающих конфигурацию некоторой программы, лучше располагать на отдельных строках, снабдив комментарием.

```
basic_colors = ("Red", "Green", "Blue")
good_style = (              # кортеж из трех элементов
               1,
               2,
               3,
             )
zero_len_tuple = ()         # задание пустого кортежа
len_1_tuple = ("a",)        # так можно задать одноэлементный кортеж
calc_tuple = ( good_style[-1]+1, good_style[-2]+2, )
```

Запись кортежа синтаксически очень похожа на запись аргументов при вызове функции. Это однако не вызывает путаницы, т. к. два операнда не могут стоять в выражении рядом, их обязательно должна связывать какая-либо операция.

Список (см. статью "Списки" в разд. 8.2.3) задается аналогично кортежу. Разница заключается в том, что в одноэлементном списке необязательно ставить запятую после элемента.

```
список: '[' [элемент[[',' элемент]+[',']]] ']'
```

Следующий пример показывает различные виды записи списков.

```
colors = ["Red", "Green", "Blue", "Violet", ]
p_languages = [            # список из трех элементов
               "Basic",
               "Pascal",
               "C",
              ]
zero_len_list = []         # задание пустого списка
len_1_list = ["a"]         # одноэлементный список
calc_list = [1, 1+2, 1+2+3]
```

Для списков подходят те же рекомендации по стилю, что и для кортежей.

В Python 2.0 появилось еще одно средство для задания списков — *списковое включение (list comprehension)*:

```
Сп_включение:  "[" выр описание "]"
описание:      вкл_for (вкл_for | вкл_if)*
вкл_for:       "for" параметр "in" последовательность
вкл_if:        "if" условие
```

Списковое включение порождает список, состоящий из элементов, вычисляемых выражением `выр`, в котором употреблены переменные, порождаемые `описанием`. Следующий пример легко проясняет ситуацию:

```
>>> print [ (a, b) for a in range(3) for b in range(4) if a > b ]
[(1, 0), (2, 0), (2, 1)]
```

В этом однострочном фрагменте сказано все, для чего раньше требовалось написать следующее:

```
tmp_list = []
for a in range(3):
  for b in range(4):
    if a > b:
      tmp_list.append( (a, b) )
print tmp_list
```

Результат один и тот же:

```
[(1, 0), (2, 0), (2, 1)]
```

Таким образом, `[ выр оп1 оп2 ... опN ]` эквивалентно коду:

```
врем_список = []
оп1:
  оп2:
    ...
      опN:
        врем_список.append( выр )
врем_список
```

Вполне элегантно выглядит задание *словаря (см. разд. 8.2.4)*. Для словарей используется третий вид скобок — фигурные скобки. Пары "ключ—значение" отделяются запятыми, а ключ от значения отделяется двоеточием.

```
словарь: "{" [key ":" val ("," key ":" val)* [","]] "}"
```

где `key` — выражение для ключа (индекса); `val` — выражение для значения.

Примеры записи словарей:

```
dict = { "1" : "один", "2" : "два", "3" : "три", }
empty_dict = {}  # пустой словарь
phonebook = {    # пример структуры данных для телефонной книги
 "Иван" : ("77777777", "ул. Большая, д. 1"),
 "Марья" : ("78787878", "ул. Малая, д. 11"),
}
address = {      # пример структуры данных для адреса
 "страна" : "Россия",
 "область" : "Республика Карелия",
 "город" : "Петрозаводск",
 "префикс" : "ул.",
 "улица" : "М.Горького",
 "корпус" : None,
 "дом" : "4",
 "квартира" : None,
}
```

Преобразование к строке

Каждый объект в Python имеет *представление (representation) в виде строки*. Окружив выражения обратными апострофами, мы получим представление результата вычисления выражений в виде строки. Напомним, что в Python

есть и другой вид преобразования к строке *(см. статью "Функции преобразования типов" в разд. 8.6.1)*. Здесь же мы рассматриваем преобразование к строке как синтаксическую конструкцию:

```
преобразование_к_строке: "`" выражение["," выражение]* "`"
```

Действие преобразования к строке аналогично применению встроенной функции `repr()`. Для преобразования вычисленных результатов вызывается метод `__repr__()` и, если выражений несколько, создается кортеж с представлением результатов в виде строк.

```
>>> print `2*2`, repr(2*2)
4 4
>>> print `type(2), "abc"`, repr((type(2), "abc"))
(<type 'int'>, 'abc') (<type 'int'>, 'abc')
>>> a = 1; b = (1, 2)
>>> print "a = "+`a`+"; b = "+`b`
a = 1; b = (1, 2)
```

8.3.2. Первичные выражения

Следующими за атомами по приоритету следуют *первичные выражения (primaries)*. Если атомы фактически дают выражению операнды, первичные выражения — наиболее приоритетные операции. К первичным выражениям относятся ссылки на атрибуты, вызовы функций, выделение срезов и взятие элементов. (Для полноты описания синтаксиса атомы тоже можно считать первичными выражениями, к которым не применена никакая операция.)

Ссылка на атрибут

Для доступа к пространству имен объекта используется хорошо известная в мире C, Pascal и объектно-ориентированного программирования нотация.

```
ссылка_на_атрибут:  первичное_выражение "." идентификатор
```

Объект, соответствующий первичному выражению, должен иметь атрибуты. Например, не все константы поддерживают ссылку на атрибут (числа, за исключением комплексных, `None` и т. п.). При отсутствии указанного идентификатором атрибута возбуждается исключение `AttributeError`. Объект пользовательского класса может перехватывать доступ к атрибутам, обрабатывая их по-своему *(см. разд. 8.4.2)*.

```
import os
print (1-1j).imag         # комплексное число
print "string".upper()    # приведение к верхнему регистру
print os.path.basename("/path/to/file.ext") # цепочка ссылок на атрибуты
```

Результат выполнения:

```
-1.0
STRING
file.ext
```

Индексные выражения

Взятие элемента последовательности или словаря (соответствия) по индексу записывается с помощью *индексного выражения* и имеет следующий синтаксис:

```
индекс_выр: первичное_выражение "[" выражение "]"
```

```
>>> lst = [1, 2, 3, 4, 5]
>>> print lst[0], lst[-1], range(100)[99]
1 5 99
>>> print ["ложь", "истина"][not lst]
ложь
```

Нумерация последовательностей начинается с `0`. Как видно из примера, отрицательные индексы отсчитывают элементы с конца. Последний элемент имеет индекс `-1`. Взятый элемент может использоваться не только для чтения его значения, но и для присваивания ему значения или удаления.

Несколько более расширенный синтаксис имеет операция *выделения среза (slicing)*. Ее потенциал не используется в стандартном Python в полном объеме и находит применение в расширении Numeric Python *(см. гл. 11)*. Для удобства рассмотрим упрощенное выделение среза:

```
индекс_выр_среза: первичное_выражение "[" [от] ":" [до] "]"
от: выражение
до: выражение
```

Наверное, будет нелишним напомнить, что в Python срез считается как бы по промежуткам между элементами, а не по самим элементам *(см. рис. 2.1 в разд. 2.4)*.

Вызовы функций и других объектов

Вызов (call) функции (или другого поддерживающего вызов объекта, *см. разд. 8.2.5*) сопровождается указанием фактических аргументов. Соответствие фактических аргументов формальным (то есть указанным в описании объекта) осуществляется как по месту в списке аргументов, так и с помощью явного указания названий формальных аргументов. Следуя документации по Python, будем различать *позиционные аргументы (positional arguments)* и *именованные аргументы (keyword arguments)*. Последние — очень удачное синтаксическое решение, ведь в случае указания имен аргументов программисту нет надобности помнить и соблюдать порядок их следования, достаточно знать лишь идентификаторы. Вызов всегда возвращает некоторое зна-

чение (если, конечно, исполнение не прерывается исключением), хотя бы None.

```
вызов:        первичное_выражение "(" [аргументы [","]] ")"
аргументы:    поз_арг_ты ["," имен_арг_ты ] | имен_арг_ты
поз_арг_ты:   выражение ("," выражение)*
имен_арг_ты: имя "=" выражение ("," имя "=" выражение)*
```

Вызов (предварительно вычисленного) объекта первичное_выражение действует следующим образом:

1. Вычисляются все выражения в скобках.
2. Устанавливается соответствие между элементами списка поз_арг_ты и первыми в определении функции формальными аргументами. Лишние позиционные аргументы заносятся в список формального аргумента "со звездочкой" ("*" имя), если таковой был задан в определении функции, иначе возбуждается исключение TypeError.
3. Устанавливается соответствие между имен_арг_ты и формальными аргументами, имеющими соответствующие имена. Если аргумент уже задействован — TypeError. Если формальный аргумент с заданным при вызове именем не найден, он заносится в словарь формального аргумента "с двумя звездочками" ("**" имя), если этот аргумент был задан, иначе возбуждается TypeError.
4. Оставшиеся формальные аргументы получают значения по заданному при определении функции умолчанию. Если значение по умолчанию не задано, возбуждается TypeError. (Значение по умолчанию для некоторого аргумента вычисляется один раз — при определении функции — и затем одно и то же значение используется при всех вызовах, если аргумент не задан.)

Формальные аргументы "со звездочками" имеют специальный смысл и не могут использоваться как аргументы напрямую, а только в качестве контейнеров для других аргументов.

Вызов различных объектов имеет разный смысл (табл. 8.18).

Таблица 8.18. *Вызов различных объектов*

Тип объекта	Реакция на вызов
Функция	Исполняется с заданными аргументами *(см. статью "Оператор def" в разд. 8.5.5)*
Встроенная функция или метод	То же; зависит от интерпретатора
Класс	Создается и возвращается новый объект класса; аргументы используются методом __init__()

Таблица 8.18 *(окончание)*

Тип объекта	Реакция на вызов
Метод объекта	Вызывается данный метод, к началу списка аргументов добавляется сам объект
Объект	Вызывается метод `__call__()` объекта с заданными аргументами (к началу списка аргументов добавлен сам объект)

8.3.3. Операции

Далее рассмотрены менее приоритетные составляющие выражений, чем атомы и первичные выражения. Именно их и принято называть операциями, хотя все вопросы касающиеся приоритета, порядка вычислений, ассоциативности уже рассмотрены ранее для более общего случая *(см. разд. 8.3)*.

Арифметические операции

В выражениях на Python доступны операции сложения, вычитания, умножения, деления, взятия остатка, возведения в степень, смены знака (унарного минуса) и унарного плюса. Вопросы их приоритета и ассоциативности уже обсуждались в начале раздела, здесь же необходимо рассмотреть, что же делают эти операции.

- *Сложение* (`"+"`) работает как по основному назначению (сложение двух чисел, приведенных к общему типу), так и для конкатенации (склеивания) последовательностей одного типа, например, строк.
- *Вычитание* (`"-"`) возвращает разность между двумя числами.
- *Умножение* (`"*"`) дает произведение двух чисел (если операнды — числа). Если же один из операндов является последовательностью, а другой — целым числом, операция возвращает повторенную заданное число раз последовательность. Неположительное число повторений дает пустую последовательность.
- *Деление* (`"/"`) также работает над числами, но при его применении следует иметь в виду, что результат оказывается не сложнее общего для операндов типа. Например, деление двух целых чисел дает целое число, которое является округлением вниз действительного числа, полученного от деления этих чисел. *Деление на ноль* возбуждает исключение `ZeroDivisionError`.

Предупреждение

В следующих версиях Python семантика деления может измениться, т. к. далеко не всех устраивает, что результат `3/2` равен `1`, а не `1.5`. По возможности, применяйте явное преобразование типа с помощью `int()`.

- *Взятие остатка от деления* ("%") первого операнда на второй осуществляется в случае, когда оба операнда — числа. Знак ненулевого результата совпадает со знаком второго операнда. Остаток от деления имеет смысл и для чисел с плавающей точкой (см. примеры ниже). Если требуется одновременно и частное, и остаток, больше подойдет встроенная функция `divmod()`. При взятии остатка от деления на ноль также возбуждается `ZeroDivisionError`.

Тем же символом "%" обозначается и операция форматирования, если первый операнд строка. Эта операция не относится к арифметическим, поэтому описана в следующем пункте.

Перед операцией числовые операнды приводятся к общему типу. В случае невозможности осуществить операцию из-за несовместимости типов возбуждается исключение `TypeError`.

Листинг 8.16. Файл arith.py. Примеры выполнения различных арифметических операций

```
examples = [ "2+2", "'2'+'2'",
             "4-2",
             "4*5", "'4'*5", "3*['a','b','c']", "'a'*'b'",
             "7/5", "7./5", "4.5/1.3", "2/0",
             "7%5", "20%4", "3.4%1.1", "7%-5", "(-7)%5", "3%0",
             "divmod(7, 5)",
             ]
# для всех примеров:
for e in examples:
  try:
    print "%-15s =" % e,
    print eval(e)
  except ZeroDivisionError, x:
    print "! ZeroDivisionError: %s" % x
  except TypeError, x:
    print "! TypeError: %s" % x
```

Результат выполнения:

```
2+2             = 4
'2'+'2'         = 22
4-2             = 2
4*5             = 20
'4'*5           = 44444
3*['a','b','c'] = ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

```
'a'*'b'          = ! TypeError: can't multiply sequence with non-int
7/5              = 1
7./5             = 1.4
4.5/1.3          = 3.46153846154
2/0              = ! ZeroDivisionError: integer division or modulo
7%5              = 2
20%4             = 0
3.4%1.1          = 0.1
7%-5             = -3
(-7)%5           = 3
3%0              = ! ZeroDivisionError: integer division or modulo
divmod(7, 5)     = (1, 2)
```

Операция форматирования

Операция форматирования принимает строку со *спецификаторами преобразований* в качестве своего первого операнда и одиночное значение любого типа, кортеж или словарь значений в качестве второго. По своему действию и даже набору спецификаций `"%"` напоминает функцию `sprintf()` языка C. Синтаксис спецификации преобразования следующий:

```
"%" [ключ][флаг*][мин_шир][.точность][длина_типа]спецификатор
ключ: "(" символ за исключением круглых скобок* ")"
флаг: "+" | "-" | пробел | "#" | "0"
мин_шир: ("1" ... "9")("0" ... "9")* | "*"
точность: ("1" ... "9")* | "*"
длина_типа: "a" ... "z" | "A" ... "Z"
спецификатор: "a" ... "z" | "A" ... "Z" | "%"
```

где `ключ` — ключ для соответствия (словаря); `флаги` — задают дополнительные свойства преобразования к строке; `мин_шир` — минимальная ширина поля; `точность` — точность (для чисел с плавающей запятой); `длина_типа` — задает модификатор типа; `спецификатор` — задает тип для представления выводимого объекта.

В табл. 8.19 приведены возможные в Python части спецификации.

Таблица 8.19. *Части спецификации формата*

Символ	Место в синтаксисе	Значение
#	флаг	Альтернативная форма представления
0	флаг	Необходимо заполнять начало поля нулями
-	флаг	Выравнивание по левому краю
+	флаг	Указание знака обязательно (даже если это "+")

Таблица 8.19 *(окончание)*

Символ	Место в синтаксисе	Значение
Пробел	`флаг`	Использование пробелов для заполнения места для знака
`*`	`мин_шир` или `точность`	Значение читается из второго операнда операции форматирования — кортежа, должно быть целого типа
`H`, `l` или `L`	`длина_типа`	В Python игнорируются
`D`, `i`	`спецификатор`	Знаковое целое
`U`	`спецификатор`	Беззнаковое целое
`O`	`спецификатор`	Восьмеричное беззнаковое целое
`x` или `X`	`спецификатор`	Шестнадцатеричное беззнаковое целое (заданное строчными или прописными латинскими буквами)
`e`, `E`	`спецификатор`	Число с плавающей точкой в формате с экспонентой (экспонента строчная или прописная соответственно)
`f`, `F`	`спецификатор`	Число с плавающей точкой
`g`, `G`	`спецификатор`	Число с плавающей точкой в более коротком написании (`e` или `f`, `E` или `F` соответственно)
`C`	`спецификатор`	Одиночный символ (целое число или односимвольная строка)
`R`	`спецификатор`	Любой объект, приведенный к строке с помощью `repr()`
`S`	`спецификатор`	Любой объект, приведенный к строке с помощью `str()`
`%`	`спецификатор`	Для задания знака процента. Для задания одиночного процента нужно записать `%%`

Предупреждение

В финансовых приложениях будьте очень внимательны, используя операцию форматирования для вывода чисел с плавающей запятой. Лучше заранее проверить правила округления положительных и отрицательных чисел, чем обнаружить ошибки в финансовых документах. Операция "`%`" использует `sprintf()` той библиотеки языка C, при помощи которой скомпилирован Python. В разных операционных системах правила округления могут различаться. Например, многие версии системы UNIX (включая GNU/Linux) используют более сложное правило округления, чем в MSC. Соответствующие примеры даны в конце этого раздела.

Наиболее типичные случаи использования операции форматирования приведены в примере ниже.

```
print """Простые примеры форматов:"""
print """Символы: '%c' и '%c'""" % (67, "k")
print """Целые: '%i', '%u', '%o', '%x', '%X'""" % tuple(range(100,105))
print """ЧПТ: '%f', '%e', '%g'""" % (0.00102345, 0.0002643, 0.0000001)
print """Объекты: '%r', '%s'""" % (xrange(5), xrange(5))
print """Более сложные примеры форматов:"""
print """Дата: %02i.%02i.%04i""" % (12, 9, 2001)
print """Дамп:""", "%02x " * 8 % tuple(range(109,117))
print """Примеры с соответствиями:"""
a, b, c = 12, 1.0/7, "Арифметический пример"
d = a + b
print """%(c)s: %(a)8.5f%(b)+8.5f=%(d)8.5f""" % vars()
print """<%(NAME)s width="%(WIDTH)i%%">%(ELEM)s</%(NAME)s>\
""" % {'NAME': 'a', 'WIDTH': 50, 'ELEM':'text'}
```

Результат выполнения:

```
Простые примеры форматов:
Символы: 'C' и 'k'
Целые: '100', '101', '146', '67', '68'
ЧПТ: '0.001023', '2.643000e-04', '1e-07'
Объекты: 'xrange(5)', 'xrange(5)'
Более сложные примеры форматов:
Дата: 12.09.2001
Дамп: 6d 6e 6f 70 71 72 73 74
Примеры с соответствиями:
Арифметический пример: 12.00000+0.14286=12.14286
<a width="50%">text</a>
```

Пример того, что *правила округления* не универсальны:

```
>>> # Linux, GCC 2.95.2
>>> print "%11.2f "*3 % (0.115, 0.125, 0.135)
       0.12        0.12        0.14
>>> # Windows, MSC или MINGW
>>> print "%11.2f "*3 % (0.115, 0.125, 0.135)
       0.12        0.13        0.14
```

В данном случае C-библиотека для Windows округляет по правилу "прибавить половину и обрубить", тогда как C-библиотека от GNU для UNIX

часть "пятерок" округляет в бóльшую сторону, а часть пятерок — в меньшую. При подсчете суммы округленных чисел этот подход библиотеки для UNIX, призванной удовлетворить ученых и инженеров, которые любят UNIX, дает меньше ошибок, однако вряд ли устроит бухгалтеров и менеджеров по продажам. Разработчикам финансового программного обеспечения можно порекомендовать самим написать правильную с их точки зрения функцию округления.

Логические операции

Python имеет *логические операции (boolean operators)*: *логическое И* (`and`), *логическое ИЛИ* (`or`) и *логическое НЕ* (`not`). Смысл этих операций должен быть понятен для всех, кто хоть немного знаком с математической логикой или языками программирования. Каждый объект в Python имеет свое *истинностное значение (truth value)* (*истина* или *ложь*). Следующие значения соответствуют лжи:

- `None`;
- числовой ноль;
- пустая последовательность (в том числе пустая строка);
- пустой словарь;
- объекты классов, метод `__nonzero__()` или `__len__()` которых возвращает ноль.

Все остальные значения считаются истинными. Проверить истинностное значение объекта можно с помощью функции `operator.truth()` *(см. разд. 9.2.6)*.

Результат операции `not` — отрицание истинностного значения аргумента: если аргумент истина, то результат `0` (ложь) и, наоборот, отрицание лжи значение `1` (истина). (`0` и `1` являются наиболее "правильными" значениями для выражения истинностного значения.) В языке Python операции `and` и `or` — не просто логические функции. Они возвращают в качестве результата один из операндов. Операнды, над которыми производятся логические операции, вычисляются по особым правилам. Правый операнд даже не вычисляется, если исход логической операции предрешен левым операндом. В операциях `and` и `or` в качестве результата всегда возвращается последний вычисленный операнд. Операция `or` возвращает истину, если истинен хотя бы один аргумент (заметьте, что это не согласуется с бытовым пониманием союза "или" и больше соответствует "и/или"). Результат операции `and` истинен, только если истинны оба аргумента. Более строго:

- Операция `not x` возвращает `1`, если `x` — истина, иначе `0`.
- Операция `x and y` возвращает `x`, если `x` является ложью, иначе `y`.
- Операция `x or y` возвращает `x`, если `x` является истиной, иначе `y`.

Логические операции лучше всего иллюстрировать с помощью *таблиц истинности*:

Листинг 8.17. Файл truth_tables.py. Печать таблиц истинности

```
print "Таблицы истинности:"
for op in ["or", "and"]:
  print "%3s" % op, 0, 1
  for A in [0, 1]:
    print "%3s" % A,
    for B in [0, 1]:
      print eval(`A`+ " " + op + " " + `B`),
    print
  print
print "not"
for A in [0, 1]:
  print "%3s" % A, not A
print
```

Результат выполнения:

```
Таблицы истинности:
 or 0 1
  0 0 1
  1 1 1

and 0 1
  0 0 0
  1 0 1

not
  0 1
  1 0
```

А следующий пример показывает логические операции в действии:

```
examples = [ "not []", "not not []", "not 'abc'", "not 4 > 5",
             "[1,2] and [2,3]", "None and 1", "2.4 and 5.8",
             "{} or 1", "'' or 'a'", "'' or []", "[] or ''",
             ]
for e in examples:
  print "%-15s ==" % e,
  print `eval(e)`
```

Результат выполнения:

```
not []           == 1
not not []       == 0
not 'abc'        == 0
not 4 > 5        == 1
[1,2] and [2,3] == [2, 3]
None and 1       == None
2.4 and 5.8      == 5.7999999999999998
{} or 1          == 1
'' or 'a'        == 'a'
'' or []         == []
[] or ''         == ''
```

Операции `and` и `or` можно в некоторых ситуациях использовать для замены условных операторов:

```
# Замена ложного значения по умолчанию:
y = x or 0      # если x -- None, то x станет равным 0
# Сохранение пустой строки и раскрытие непустой строки:
y = x and ("""<font color="Green">%s</font>""" % x)
# Выбор одной из двух альтернатив (не универсален!):
y = x and "true" or "false"
# Выбор одного из вариантов (не универсален!):
y = (x < 0 and "<") or (x == 0 and "=") or ">"
```

Заметьте, что последние два выражения корректны только потому, что в качестве результатов используются непустые строки, т. е. значения истинные (`"true"`, `"false"`, `"<"`, `"="`, `">"`). Применяйте подобные конструкции только там, где истинность значений результатов гарантирована.

Сравнения

В языке Python доступны как обычные *операции сравнения* величин, такие как `<` *(меньше)*, `>` *(больше)*, `<=` *(меньше или равно)*, `>=` *(больше или равно)*, `==` *(равно)*, `!=`, `<>` *(не равно)*, так и некоторые другие: `is` *(тождественно равно (is))*, `is not` *(тождественно не равно (is not))*, `in` *(принадлежит (in))*, `not in` *(не принадлежит (not in))*. Кроме `in` и `not in`, которые определены только для последовательностей, остальные операции действуют для всех объектов.

В случае успеха операция (или цепочки операций) сравнения возвращает `1` ("истина"), в противном случае — `0` ("ложь").

В отличие от других языков программирования, в Python сравнения можно записывать в виде цепочки, как это принято в математике. Как уже говорилось ранее, цепочка сравнений вида `a op1 b op2 c ... y opn z` фактиче-

ски является сокращенной записью для `(a op1 b) and (b op2 c) and ... and (y opn z)`, где `op1, ..., opn` — любые операции сравнения.

Листинг 8.18. Режим буднего дня

```
for i in range(0, 24):
  if 9 <= i < 13 or 14 <= i <= 18:  # рабочие часы
    print "р",
  elif i == 13:                     # перерыв
    print "п",
  elif i < 8:                       # сон
    print "с",
  else:                             # остальное время
    print "о",
```

Результат выполнения:

```
с с с с с с с с о р р р р п р р р р р о о о о о
```

О сравнении объектов одного типа легче говорить, объяснив правила их упорядочивания. Целые, длинные целые и числа с плавающей точкой упорядочиваются по принятым в математике правилам. Комплексные числа сортируются по действительным частям и, если те совпадают, по мнимым. Для сравнения числа сначала приводятся к общему типу. Символы упорядочиваются по коду. Строки принято упорядочивать в *лексикографическом порядке*. Они сравниваются от начала, посимвольно до первого несовпадения символов, после чего исход сравнения строк зависит от результата сравнения этих символов. Если одна строка кончилась раньше, чем не совпали символы, она стоит раньше по порядку. На основе таких же правил упорядочиваются и другие последовательности, только вместо символов используются элементы последовательности. Словари в Python упорядочиваются как списки пар `(ключ, значение)`. Сравнение пользовательских объектов обеспечивает метод `__cmp__()` *(см. разд. 8.4.3)*. Для всех остальных типов естественного и интуитивно понятного порядка может не существовать. Объекты таких типов все равно будут сравниваться одинаково в пределах одного запуска программы, но результат их сравнения не имеет практического смысла. Также не имеет смысла сравнивать разнородные объекты.

```
print "Удачные сравнения (результаты -- истины):"
print 0 < 1L < 2.0 < 3-1j < 3 < 3+1j,
print "беспорядок" < "порядок" < "порядок лексикографический",
print "Порядок" != "порядок",
print (1, 1, 1) < (1, 1, 1000) < (2, 1, 1) < (2, 1, 1, -1)
print "Неудачные сравнения:"
```

```
print 2 < 3 < "4" < 5,
print 1 < 1.0000000000000001
```

Результат выполнения:

```
Удачные сравнения (результаты -- истины):
1 1 1 1
Неудачные сравнения:
0 0
```

Операция `is` проверяет тождественность двух объектов. Она возвращает истину только если некоторый объект сравнивается с самим собой. Строки и числа одного типа с одинаковым значением считаются одинаковыми объектами. Операция `is not` является отрицанием `is`, т. е. `(a) is not (b)` равносильно `not (a) is (b)`.

```
a = [1,2]       # заводим список и даем ему имя a
b = a           # даем списку второе имя -- b
c = a[:]        # копии списка даем имя c
print "1" is "1", 1 is 1, None is None, a is b
print 1 is not 1.0, [1,2] is not [1,2], a is not c, a is not [1,2]
```

Результат выполнения:

```
1 1 1 1
1 1 1 1
```

Операция `in` определена только в случае, если второй операнд — последовательность (иначе возбуждается `TypeError`). Выражение `x in s` дает `1` ("истина"), если в `s` существует элемент `y == x`. Операция `(x) not in (s)` равносильна `not (x) in (s)`.

Листинг 8.19. Программа, сообщающая о праздничных днях и каникулах

```
import time
holydays = [(1, 1), (1, 7), (3, 8), (5, 1), (5, 9), (6, 12), (10, 7)]
(_,month,day,_,_,_,_,_,_) = time.localtime(time.time())
if (month,day) in holydays:
  print "Сегодня праздник"
if (1, 1) <= (month,day) <= (1, 15) or (6, 15) <= (month,day) <= (9, 15):
  print "Каникулы"
```

Побитовые операции

Все данные в компьютере представлены, в конечном счете, в виде двоичных разрядов, собранных для удобства обработки в байты и более длинные последовательности. В некоторых случаях даже в языке сверхвысокого уровня при-

ходится работать с данными на битовом уровне. Кроме того, некоторые стандартные библиотеки используют битовое представление для передачи флагов или просто для более плотного хранения данных. Ниже мы не только рассмотрим побитовые операции, но и обсудим их практическое применение.

В Python определены следующие побитовые операции: `|` — *побитовое ИЛИ*, `^` — *побитовое исключающее ИЛИ*, `&` — *побитовое И*, `<<` — *сдвиг влево*, `>>` — *сдвиг вправо*, `~x` — *побитовое НЕ* (*побитовая инверсия*). Операндами может служить целое или длинное целое. Остальные типы возбуждают исключение `TypeError`.

```
def pr(x):
  """красивая печать операции и ее результата"""
  print "%15s == 0x%08x" % (x, eval(x))
for op in "|^&":
  pr("0x1100 "+op+" 0x1010")
pr("~ 0x1010")
pr("0x0010 << 1")
pr("0x0010 >> 1")
```

Результаты побитовых операций лучше смотрятся в двоичной или шестнадцатеричной системах счисления:

```
0x1100 | 0x1010 == 0x00001110
0x1100 ^ 0x1010 == 0x00000110
0x1100 & 0x1010 == 0x00001000
       ~ 0x1010 == 0xffffefef
    0x0010 << 1 == 0x00000020
    0x0010 >> 1 == 0x00000008
```

Замечание

Операции сдвига почти равносильны делению (сдвиг вправо) или умножению (сдвиг влево) на степень двойки. Почти — потому что отрицательные числа хранятся в инвертированном виде, а старший бит обозначает знак числа. Вот и получается, что сколько `-1` (в двоичном представлении это — `11111111111111111111111111111111`) ни сдвигай вправо — она все равно останется `-1`.

В следующем примере операция смены знака моделируется с помощью сложения и побитового НЕ (напомним, что побитовое НЕ имеет бóльший приоритет), а логическая операция `xor` (исключающее ИЛИ) через побитовое исключающее ИЛИ:

```
def neg(x):
  """Смена знака числа"""
  return ~x+1
```

```
def xor(x, y):
  """Логическое исключающее ИЛИ"""
  return (not x) ^ (not y)
  # операции not нужны для приведения значений к 0 и 1
  # (на xor не влияет одновременное отрицание обоих аргументов)

print neg(4), neg(0), neg(-3)
print xor(1, 0), xor(0, 0), xor([], 1), xor(1, [1])
```

Результат выполнения:

```
-4 0 3
1 0 1 0
```

Но основное применение побитовых операций — запись и чтение флагов. Под *флагом* здесь понимается признак, который либо присутствует, либо отсутствует. Установленному флагу соответствует `1`, сброшенному — `0`. Таким образом, в одном байте можно хранить сразу восемь флагов! В приведенном ниже примере с помощью флагов меняется поведение функции `re.search()` из модуля `re`.

```
>>> import re
>>> text = "abc Abcd ABCDE"
>>> print re.search("ABC", text).span()
(9, 12)
>>> print re.search("ABC", text, re.IGNORECASE).span()
(0, 3)
>>> print re.search("A B C", text, re.VERBOSE).span()
(9, 12)
>>> print re.search("A B C", text, re.IGNORECASE|re.VERBOSE).span()
(0, 3)
>>> print hex(re.IGNORECASE), hex(re.VERBOSE)
0x2 0x40
```

В этом примере третий аргумент функции `re.search()` — набор флагов, собранных вместе с помощью побитового ИЛИ. Флаг `re.IGNORECASE` устанавливает второй бит, а `re.VERBOSE` — седьмой. Комбинация этих флагов (`0x42`) имеет оба этих бита. В исходных текстах модуля `re` (в Python 2.0 — это файл sre_compile.py), мы можем увидеть код для проверки флага (`flags` — комбинация флагов, `SRE_FLAG_IGNORECASE == re.IGNORECASE`):

```
if flags & SRE_FLAG_IGNORECASE:
      emit(OPCODES[OP_IGNORE[op]])
   else:
      emit(OPCODES[op])
```

Таким образом, проверить установленность флага можно с помощью побитового И.

Побитовые операции также активно применяются в конверторах бинарных форматов, например, при кодировании в формат base64, и вообще при разборе бинарных файлов.

8.3.4. Лямбда-выражение

С помощью *лямбда-выражений* можно определять безымянные функции. В отличие от оператора `def` определение функции может состоять лишь из одного выражения, которое и вычисляется при вызове функции.

```
lambda_form:    "lambda" [список_аргументов]: выражение
```

где `список_аргументов` — список формальных аргументов функции (возможно, с заданными значениями по умолчанию); `выражение` — выражение, определяющее функцию.

Основные потребители безымянных функций — функции `map()`, `filter()` и `reduce()` *(см. статью "Функции обработки данных" в разд. 8.6.1)*.

```
print reduce(lambda x, y: x+y, [1,2,3])       # Сумма.
print filter(lambda x: x%12==0, range(100))   # Числа кратные на 12.
print map(lambda x,y: x%y,                    # Применение
          ["%i", "%03i", "%-4i"],             # форматов
          [133, 2, 123])                      # к числам.
```

Результат выполнения:

```
6
[0, 12, 24, 36, 48, 60, 72, 84, 96]
['133', '002', '123 ']
```

Внутри `выражения` не видны локальные переменные (только глобальные), поэтому для передачи локальных значений используется механизм для задания значений по умолчанию (в `списке_аргументов` локальные имена доступны). При этом формальные аргументы могут иметь имена фактических аргументов — локальных переменных:

```
def linear(v1, v2, a, b=1):
  """Линейная комбинация векторов a*v1 + b*v2"""
  return map(lambda x, y, a=a, b=b: a*x + b*y, v1, v2)

print linear([1, 3, -1], [1, 0, 1], -1, 2)
```

Результат выполнения:

```
[1, -3, 3]
```

В Python 2.1 и выше можно не писать `a=a`, `b=b`, т. к. локальные имена объемлющего блока будут видны внутри лямбда-выражения.

8.4. Специальные имена методов

В этом разделе читатель познакомится с возможностями Python по *эмуляции* встроенных объектов в классах, определяемых пользователем. При этом программисту доступна вся мощь синтаксической поддержки встроенных классов. Достигается это определением в классе методов со специальными именами (начинающимися и оканчивающимися на два подчеркивания).

Этот раздел не совсем обычен, т. к. описывает не готовые функции или методы, а только интерфейсы, которые нужно поддерживать, чтобы объекты пользовательского класса могли играть определенные *роли* встроенных объектов, а, значит, пользоваться уже написанными функциями Python или его стандартных модулей. Для каждой операции (встроенные здесь не исключение) требуется строго определенный набор методов и поведение (то есть роль). Например, чтобы объект мог работать в роли последовательности в цикле `for`, он должен иметь метод `__getitem__()` и поднимать исключение `IndexError` в случае неудачи взятия элемента.

Особо следует подчеркнуть, что для эмуляции бинарных операций обычно приходится определять два метода (назовем их `__op__()` и `__rop__()`), первый из которых применяется, если объект стоит слева от операции, а другой для случая, когда объект справа от знака операции. Предпочтение отдается методу `__op__()`.

8.4.1. Базовая настройка

Далее будут описаны основные методы, которые вызываются при создании, удалении, сравнении, представлении объекта в виде строки, а также при использовании объекта в качестве логического значения.

Инициализация объекта: метод *__init__()*

Метод `__init__()` вызывается при создании объекта и служит для его инициализации. Метод получает в качестве аргумента уже созданный объект (`self`) и аргументы, которые были заданы при вызове конструктора класса.

```
__init__ (self[, аргументы...])
```

Внутри метода `__init__()` нужно явно вызывать инициализацию базового класса. Это можно сделать, вызвав `базовый_класс.__init__()` со всеми необходимыми аргументами.

```
class BASE:                          # базовый (для ACLASS) класс
  def __init__(self, b):             # инициализация BASE
```

```
    self.basevalue = b
  def give(self):
    return (self.basevalue,)

class ACLASS(BASE):
  def __init__(self, b, a):      # инициализация ACLASS
    BASE.__init__(self, b)       # вызов инициализации BASE
    self.avalue = a
  def give(self):
    return self.basevalue, self.avalue

a = ACLASS("1", "2")
b = BASE("3")

print a.give(), b.give()
```

Результат выполнения:

```
('1', '2') ('3',)
```

Деструктор объекта: метод *__del__()*

Метод `__del__()` вызывается при удалении объекта из памяти *(см. статью "Оператор del" в разд. 8.5.1)*, т. е. когда количество ссылок на объект стало равным нулю. По аналогии с `__init__()` деструктор может явно вызывать деструктор базового класса, если это необходимо.

```
__del__ (self)
```

Деструктор нужен только в том случае, если объект управляет некоторым внешним (по отношению к программе) ресурсом, который надо привести в определенное состояние. Например, если объект управляет сеансом работы по модему, то деструктор должен положить трубку и, может быть, проинициализировать модем, приведя его в состояние готовности. В отличие от C, Python сам удаляет объекты, ставшие ненужными, поэтому деструктору незачем заботиться об освобождении памяти и тому подобных вещах.

Предупреждение

Исключения, происходящие в методе `__del__()`, игнорируются. Предупреждение об исключении печатается на `sys.stderr`. Кроме того, нужно стараться делать деструкторы как можно более краткими, не использующими ничего лишнего, т. к. они могут исполняться в условиях, когда глобальное пространство имен заметно поредело и другие объекты недоступны.

```
class A:
  def __del__(self):
    print "Удаляют!"

a = A()
print "Программа кончилась."
```

Результат выполнения:

```
Программа кончилась.
Удаляют!
```

Никогда не пишите деструкторы "ради проформы": экземпляры классов, у которых имеется метод `__del__()`, не могут быть освобождены из памяти автоматическим механизмом сборки мусора. Поэтому нужно особенно тщательно следить, чтобы подобные объекты не оказались в цепочке циклических ссылок *(см. разд. 9.2.4)*.

Представление объекта в виде строки: методы *__repr__()* и *__str__()*

Методы `__repr__()` и `__str__()` служат для представления объекта в виде строки, в функциях `repr()` (равно как в обратных апострофах) и в функции `str()` (и в операторе `print`). В документации к Python пояснено, что `__repr__()` должен по возможности давать такое представление объекта, при котором он может быть легко воссоздан простым вычислением полученной строки, либо информация об объекте должна быть заключена в угловые скобки. `__str__()` дает более неформальное, ориентированное на человека, представление объекта.

- `__repr__(self) -> string`
- `__str__(self) -> string`

Листинг 8.20. Класс `point` и представление его экземпляров в виде строки

```
class point:
  def __init__(self, x, y):
    self.coord = (x, y)
  def __repr__(self):
    return "point(%s, %s)" % self.coord
  def __str__(self):
    return "(%s, %s)" % self.coord
```

```
P = point(2, 3.4)
print P, str(P)
print `P`, repr(P)
```

Результат выполнения:

```
(2, 3.4) (2, 3.4)
point(2, 3.4) point(2, 3.4)
```

Сравнение объектов

Методы `__cmp__()` и `__rcmp__()` используются операциями сравнения и сортировки.

- `__cmp__ (self, other) -> -1, 0, 1`
- `__rcmp__ (self, other) -> -1, 0, 1`

Методы должны возвращать `0`, если объекты равны, отрицательное целое, если `self < other` и положительное целое в случае `self > other`.

О функции `cmp()` *см. статью "Числовые и символьные функции" в разд. 8.6.1.*

Следующий пример помогает установить отношение порядка на объектах `point` (точка) на плоскости.

Листинг 8.21. Файл cmp_points.py. Точки на плоскости. Сравнение

```
class point:
  def __init__(self, x, y):
    self.coord = (x, y)
  def __repr__(self):
    return "point(%s, %s)" % self.coord
  def __str__(self):
    return "(%s, %s)" % self.coord
  def _r(self):
    """Квадрат расстояния до нуля"""
    return self.coord[0]**2+self.coord[1]**2
  def __cmp__(self, dr):
    """Сравнение точек по расстоянию до нуля"""
    return self._r() - dr._r()
  __rcmp__ = __cmp__              # в данном случае методы одинаковы

print point(0,10) > point(5,5) > point(6, 2)
lst = [point(0,10), point(5,5), point(6, 2), point(0,0)]
lst.sort()
print lst
```

Результат выполнения:

```
1
[point(0, 0), point(6, 2), point(5, 5), point(0, 10)]
```

В Python 2.1 можно эмулировать сравнения по-отдельности. Для этого можно определить методы `__lt__()` (меньше), `__gt__()` (больше), `__le__()` (меньше или равно), `__ge__()` (больше или равно), `__ne__()` (не равно), `__eq__()` (равно). Таким образом, численные приложения Python получили возможность использовать интуитивный синтаксис для операций поэлементного сравнения объектов.

Хэш-метод *__hash__()*

Метод `__hash__()` нужен для использования объекта в качестве ключа в словарях, а также для других применений, требующих вызова встроенной функции `hash()`. Данный метод бесполезен, если класс также не определяет рассмотренный ранее метод `__cmp__()`. *Хэш-метод* — это фактически *хэш-функция* от объекта, поэтому она должна удовлетворять некоторым правилам:

- значения хэш-функции от объектов с одинаковым (согласно `__cmp__()`) значением должны совпадать;
- хэш-функция должна зависеть некоторым образом от состояния объекта (то есть от значений всех его атрибутов);
- хэш-функция должна равномерно "перемешивать" объекты, чтобы возникало как можно меньше коллизий из-за одинаковых значений хэш-функции.

Хэш-функция используется следующим образом.

```
__hash__(self) -> int
```

Листинг 8.22. Использование пользовательского класса в качестве ключей словаря

```
class obj2:
  """Класс, объекты которого могут служить ключами словаря"""
  def __init__(self, x):
    self.value = x
  def __hash__(self):
    return hash(self.value)
  def __cmp__(x, y):
    return cmp(x.value, y.value)

o2 = obj2(0)
d = {o2 : 0, obj2(1) : '2', obj2(2) : '3'}   # создаем словарь
```

```
print obj2(2)==obj2(2)   # объекты "равны"
print d[o2], d[obj2(2)]
```

Результат выполнения:

```
1
0 3
```

(См. также разд. 8.2.4.)

Проверка на истинность: метод *__nonzero__()*

Метод `__nonzero__()` необходим для определения логического значения объекта. Он должен возвращать `1`, если в логических операциях объект нужно считать истиной, и `0` — если наоборот. Если метод `__nonzero__()` неопределен, используется метод `__len__()`, описанный далее в данном разделе. Если и этот объект не определен, объект считается истиной.

```
__nonzero__ (self) -> 0, 1
```

Наш пример с точками на плоскости поможет проиллюстрировать действие этого метода:

```
class point:
  def __init__(self, x, y):
    self.coord = (x, y)
  def __nonzero__(self):
    return self.coord[0] != 0 or self.coord[1] != 0


for x in range(-4, 4):
  for y in range(-4, 4):
    if point(x,y):
      print ".",
    else:
      print "*",
  print
```

Результат:

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . * . . .
. . . . . .
. . . . . .
. . . . . .
```

8.4.2. Настройка доступа к атрибутам

Простой доступ к атрибутам объектов Python возможен и без дополнительных методов *(см. статью "Ссылка на атрибут" в разд. 8.3.2)*, однако в некоторых случаях программисту нужно вмешаться в этот механизм и сделать все по-своему. Для этого он должен определить хотя бы один из следующих методов: `__getattr__()`, `__setattr__()`, `__delattr__()`.

```
class P:
  class_attribute = "ca"
  def __getattr__(self, attname):
    print "Getting", attname
  def __setattr__(self, attname, value):
    print "Setting", attname, "to", value
  def __delattr__(self, attname):
    print "Deleting", attname


P().aaaaaaaaa
P().bbbbbbbbb = 2
print P().bbbbbbbbb
del P().zzzzzzz
print P().class_attribute
```

Результат выполнения:

```
Getting aaaaaaaaa
Setting bbbbbbbbb to 2
Deleting zzzzzzz
ca
```

Чтение атрибута: метод *__getattr__()*

Метод `__getattr__()` вызывается в том случае, если запрашиваемый атрибут не находится обычным порядком (то есть такого атрибута нет ни у самого объекта, ни у его класса и всех классов, от которых данный класс унаследован).

Листинг 8.23. Пример, в котором доступ к атрибутам заменяется вызовом методов

```
class Smart:
  def _f_top(self):
    print "(TOP)"
    return 1
```

```
  def _f_bottom(self):
    print "(BOTTOM)"
    return 0
  def __getattr__(self, attname):
    return eval("self._f_%s()" % attname)

s = Smart()
print s.top, s.bottom
```

Примечание

В Python 2.2 появится специальный атрибут `__getattribute__()`, который будет вызываться во всех случаях, даже если атрибут существует.

Запись атрибута: метод *__setattr__()*

Метод `__setattr__()` вызывается когда атрибуту пытаются присвоить некоторое значение. В отличие от `__getattr__()`, `__setattr__()` вызывается всегда, поэтому внутри него нельзя использовать доступ к атрибуту. Вместо `self.attname = value` нужно записывать значения прямо в словарь объекта: `self.__dict__[attname] = value`.

Листинг 8.24. Пример, в котором перехватывается запись атрибутов

```
class Smart:
  allowed = ["a", "b", "c"]        # разрешенные атрибуты
  def __setattr__(self, attname, value):
    if attname in self.allowed:    # если атрибут среди разрешенных:
      self.__dict__[attname] = value     # присваиваем ему значение
    else:                          # иначе возбуждаем исключение
      raise AttributeError, "Setting %s is not allowed" % attname

s = Smart()
s.a = 1
s.c = 3
print s.a, s.c
s.d = 4
```

Результат выполнения:

```
1 3
Traceback (most recent call last):
```

```
  File "<stdin>", line 13, in ?
  File "<stdin>", line 7, in __setattr__
AttributeError: Setting d not allowed
```

Удаление атрибута: метод *__delattr__()*

Метод `__delattr__()` вызывается при удалении атрибута оператором `del`. Здесь следует отметить, что далеко не для всех объектов подобная операция имеет смысл.

```
class Dumb:
  def __delattr__(self, attname):
    print "eh?"


a = Dumb()
del a.someattribute
```

8.4.3. Эмуляция числовых типов

Все операции над числами и многие встроенные функции могут работать не только над встроенными числовыми типами. Любой класс можно заставить вести себя как число и полностью интегрироваться с существующими типами числовых данных. Будет нелишним напомнить, что приоритеты операций *(см. разд. 8.3)* изменить нельзя: они остаются те же, что и у встроенных операций.

Примечание

Для большей краткости изложения ниже мы отошли от традиции называть первый аргумент метода именем `self`.

Методы для арифметических операций

Методы `__add__(x, y)`, `__radd__(y, x)`, `__sub__(x, y)`, `__rsub__(y, x)`, `__mul__(x, y)`, `__rmul__(y, x)`, `__div__(x, y)`, `__rdiv__(y, x)`, `__mod__(x, y)`, `__rmod__(y, x)`, `__divmod__(x, y)`, `__rdivmod__(y, x)` соответствуют операциям сложения `x+y`, вычитания `x-y`, умножения `x*y`, деления `x/y`, взятия остатка от деления `x%y` и встроенной функции `divmod(x,y)`.

Конечно, совсем необязательно, чтобы объект, у которого определены указанные выше методы, был числом. Для читаемости программы более важно, чтобы символ операции соответствовал ее смыслу.

Методы `__pow__(x, y[, m])` и `__rpow__(y, x)` соответствуют операции возведения в степень `x**y` и аналогичной встроенной функции

`pow(x,y[,m])`. Третий аргумент необязателен *(см. статью "Числовые и символьные функции" в разд. 8.6.1)*.

В Python 2.0 была добавлена возможность расширенного присваивания:

```
>>> a = 5
>>> a *= 2 + 2
>>> print a
20
```

Метод для присваивания с соответствующей операцией имеет имя `__iop__()`, где `op` — бинарная операция, поддерживающая расширенное присваивание. Вот эти методы: `__iadd__(x, y)`, `__isub__(x, y)`, `__imul__(x, y)`, `__idiv__(x, y)`, `__imod__(x, y)`, `__ipow__(y, x)`. Интересно, что инкрементные присваивания будут работать и без этих методов, применяя операцию и присваивание раздельно. Для нетривиальных объектов это может быть очень неэффективно.

Методы `__neg__(x)` и `__pos__(x)` служат для реализации унарных операций смены знака `-` и позитива `+`, а метод `__abs__(x)` соответствует функции `abs`.

В следующем примере определяется класс `Tag` для моделирования тэгов языка HTML.

Листинг 8.25. Файл tag_class.py. Класс `Tag`

```
class Tag:
  """Простейшая модель парного тэга"""
  def __init__(self, _name, **_atts):
    """Инициализируем значениями именованных аргументов из
    специального аргумента **_atts"""
    self._name = _name
    self._atts = _atts   # словарь имен аргументов и их значений
  def _put_atts(self):   # печать атрибутов
    atts = self._atts.items()
    atts.sort()
    r = ""
    for att in atts:
      r = r + ' %s="%s"' % att
    return r
  def __mod__(self, data):   # пишем data внутрь тэга
    return '<%s%s>%s</%s>' % (self._name, self._put_atts(),
                              data, self._name)
```

```
t = Tag("a", href="http://www.python.org", name="PYTH")
i = Tag("i")
print t % (i % "Сайт по Python")
```

Результат выполнения:

```
<a href="http://www.python.org" name="PYTH"><i>Сайт по Python</i></a>
```

Методы для побитовых операций

Методы `__lshift__(x, y)`, `__rlshift__(y, x)`, `__rshift__(x, y)`, `__rrshift__(y, x)`, `__and__(x, y)`, `__rand__(y, x)`, `__or__(x, y)`, `__ror__(y, x)`, `__xor__(x, y)`, `__rxor__(y, x)` соответствуют побитовым операциям сдвигов влево и вправо (`<<` и `>>`), побитовых И, ИЛИ и исключающего ИЛИ (`&`, `|` и `^`).

Аналоги тех же методов для расширенного присваивания: `__ilshift__(x, y)`, `__irshift__(x, y)`, `__iand__(x, y)`, `__ior__(x, y)` и `__ixor__(x, y)`. *(см. статью "Оператор присваивания" в разд. 8.5.1).*

Для реализации унарной операции побитовой инверсии ("`~`") нужно определить метод `__invert__()`.

Для встроенных логических операций `and`, `or` и `not` определение методов не предусмотрено т. к. они действуют по правилам, универсальным для всех объектов. От объектов требуется только предоставлять свое логическое значение *(см. статью "Проверка на истинность: метод __nonzero__()" в разд. 8.4.1).*

Методы для организации преобразований типов

Методы `__int__(x)`, `__long__(x)`, `__float__(x)`, `__complex__(x)` должны преобразовывать объект к соответствующему числовому типу (целому, длинному целому, с плавающей точкой, комплексному). Данные преобразования используются и в операции форматирования "`%`" для преобразования объекта к нужному числовому типу.

Метод `__coerce__(x, y)` играет очень важную роль в реализации нового числового типа. Он служит для согласования типов операндов, поступающих на вход бинарных операций.

`__coerce__(x, y) -> (x1, y1)` или `None`

Данный метод должен преобразовывать свои аргументы `x`, `y` в кортеж из двух элементов `(x1, y1)` одного типа, либо возвращать `None`, если преобразование невозможно. Если один объект-операнд не смог преобразовать оба операнда к общему типу, будет вызван `__coerce__()` другого операнда. До-

кументация по языку Python рекомендует возвращать из `__coerce__()` для первого операнда `None`, если оба операнда нужно приводить к типу второго операнда (объекту "легче" преобразовывать к своему типу, т. к. обычно общий для обоих операндов тип является более сложным).

```
class scalar:
  """Класс скаляр"""
  def __init__(self, v):            # инициализируем
    self.value = v
  def __add__(x, y):                # определяем сложение
    return scalar(x.value+y.value)
  __radd__ = __add__                # то же
  def __coerce__(x, y):             # приведение к типу скаляр
    if type(y) == type(1) or type(y) == type(1.2):
      return (x, scalar(y))         # числа приводим к типу скаляр
    if y.__class__.__name__ == "scalar":
      return (x, y)                 # здесь ничего не меняем
    # в других случаях -- None
  def __str__(self):                # представление объекта
    return "scalar(%f)" % self.value

print scalar(2.3)+2, scalar(3)+scalar(4), 4.5+scalar(8.1)
```

Результат выполнения:

```
scalar(4.300000) scalar(7.000000) scalar(12.600000)
```

Методы *__oct__()* и *__hex__()*

Методы `__oct__(x)` и `__hex__(x)` предназначены для задания восьмеричного и шестнадцатеричного представления объекта.

8.4.4. Эмуляция последовательностей и отображений

Последовательности и отображения (словари) имеют много различных методов, часть из которых закреплена синтаксически. Чтобы построить объект, ведущий себя как последовательность, необходимо определить несколько методов (табл. 8.20). Кроме того, набор методов неизменяемых и изменяемых последовательностей несколько отличается, а словари и изменяемые последовательности имеют несколько общих методов.

Таблица 8.20. *Методы, рекомендуемые для построения различных типов данных*

Метод	Изменяемая последовательность	Неизменяемая последовательность	Словарь
`__len__()`	+	+	+
`__getitem__()`	+	+	+
`__setitem__, __delitem__()`	+	-	+
`__getslice__()`	+	+	+
`__setslice__(), __delslice__()`	+	-	-
`__add__(), __radd__(), __iadd__(), __mul__(), __rmul__(), __imul__()`	+	+	-
`append(), count(), index(), insert(), pop(), remove(), reverse(), sort()`	+	-	-
`keys(), values(), items(), has_key(), get(), clear(), copy(), update()`	-	-	+

(См. разд. 8.2.3 и *разд. 8.2.4.)*

Рассмотрим методы по порядку.

Метод *__len__()*

Метод `__len__()` должен возвращать длину объекта (количество символов, элементов, пар ключей или т. п.). Если этот метод возвращает `0` и нет метода `__nonzero__()`, значение, даваемое методом, используется для определения логического значения объекта: объект ненулевой длины имеет логическое значение "истина", а нулевой — "ложь".

Метод *__getitem__()*

Метод `__getitem__()` реализует возможность взятия элемента или выделения среза.

```
__getitem__ (self, key) -> value
```

Последовательности принимают в качестве `key` целые значения или объекты-срезы, а словари — объекты, имеющие метод `__hash__()`. В Python принято, чтобы отрицательные индексы соответствовали отсчетам от конца последовательности. Об этом метод `__getitem__()` должен позаботиться. Если `key` неподходящего типа, нужно возбудить исключение `TypeError`, в противном

случае подходит значение ключа — `KeyError` (для словарей), `IndexError` (для последовательностей) или (более общее) `LookupError`.

Листинг 8.26. Организация доступа к файлу как к неизменяемой последовательности

```
class file_as_array:
  """Файл как массив символов"""
  def __init__(self, filename):
    """Инициализация состоит в открытии файла"""
    self.f = open(filename, "rb")
    self.f.seek(0, 2)             # на конец файла
    self.filesize = self.f.tell() # получаем размер файла
  def __len__(self):
    """Длина файла"""
    return self.filesize
  def __del__(self):
    """При удалении объекта закрываем файл"""
    self.f.close()
  def __getitem__(self, i):
    """Взятие куска файла"""
    from types import SliceType, IntType, LongType
    if type(i) == SliceType:           # если задан срез:
      start, stop = i.start, min(i.stop, len(self))
    elif type(i) in [IntType, LongType]: # если задан индекс:
      start, stop = i, i+1
    else:
      raise TypeError, "Неподходящий тип индекса"
    bytes = stop-start
    # переход к неотрицательным индексам:
    if start < 0:
      start += len(self)
    if stop < 0:
      stop += len(self)
    if not (0 <= start < len(self) and 0 <= stop <= len(self)):
      raise IndexError, "За пределами файла"
    self.f.seek(start, 0)
    return self.f.read(bytes)
```

```
fa = file_as_array("res.tmp")
print len(fa)
print `fa[1]`, `fa[0:6]`
print `fa[-3:]`, `fa[-2]`
```

Результат выполнения:

```
20791
'H' '<HTML>'
'txt' 'x'
```

О том, что должен делать метод `__getitem__()`, чтобы объект можно было ставить в качестве последовательности в цикле `for` *(см. статью "Оператор for" в разд. 8.5.2)*.

Метод *__setitem__()*

Метод `__setitem__()` призван присваивать значение элементу словаря или элементам последовательности по заданному индексу или слою.

```
__setitem__ (self, key, value)
```

Требование к `key` те же, что и для `__getitem__()` *(см.* выше).

Следующий пример взят из исходного текста модуля `UserString`, из класса `MutableString`, сделанного специально для тех, кому нравятся строки а-ля Basic (то есть изменяемые):

```
def __setitem__(self, index, sub):
    if index < 0 or index >= len(self.data): raise IndexError
    self.data = self.data[:index] + sub + self.data[index+1:]
```

Метод *__delitem__()*

Метод `__delitem__()` должен удалять элемент по указанному индексу или слою.

```
__delitem__ (self, key)
```

Требование к `key` те же, что и для `__getitem__()` и `__setitem__()` *(см.* выше).

В Python 2.0 описанные `__getitem__()`, `__setitem__()`, `__delitem__()` работают с объектами-срезами. До этой версии в Python использовались другие методы для работы со срезами. О них речь пойдет ниже.

Методы *__getslice__(), __setslice__()* и *__delslice__()*

Раньше, когда в Python не было расширенной поддержки срезов, и они задавались как `[i:j]`, для выделения срезов использовались методы `__getslice__()`, `__setslice__()` и `__delslice__()`.

❐ `__getslice__ (self, от, до)`

- `__setslice__ (self, от, до, seq)`
- `__delslice__ (self, от, до)`

Так, например, `a[2:4] = [1]` приводило к вызову `__setslice__(a, 2, 4, [1])`.

Если указанные методы (начиная с Python 2.0 они считаются устаревшими) отсутствуют, интерпретатор формирует объект-срез и передает его методам `__getitem__()`, `__setitem__()` или `__delitem__()` соответственно.

8.4.5. Поддержка вызова

Любой определенный пользователем объект в Python можно сделать поддерживающим вызов аналогично функции. Для этого в нем нужно определить метод `__call__()`.

```
__call__(self[, аргументы])
```

После этого объект можно вызывать как функцию:

```
obj([аргументы])
```

При этом на самом деле будет вызываться:

```
obj.__call__([аргументы])
```

Листинг 8.27. Пример объекта, поддерживающего вызов

```
class CallMe:
  def __init__(self, x):
    self.x = x
  def __call__(self, delta=0):
    return self.x + delta
c = CallMe(5)
print c(3)
```

Результат выполнения:

```
8
```

8.5. Операторы

Операторы (statements) — записанные в тексте программы предписания интерпретатору. Оператор обычно соответствует нескольким низкоуровневым командам платформо-независимого байт-кода Python *(см. примеры в разд. 9.18.3)*. Команда байт-кода, в свою очередь, вызывает при интерпретации выполнение сотен машинных команд для данного типа процессора.

Операторы бывают *простые (simple)* и *составные (compound)*. Простой оператор помещается на одной логической строке, тогда как составной имеет в своем составе *фрагмент кода*. Кроме того, составные операторы могут иметь несколько частей и, соответственно, несколько фрагментов кода, например:

```
if a < b:
  a, b = b - a, a
  print "a < b"
else:
  a, b = b, a - b
  print "b <= a"
```

Фрагменты, располагающиеся на отдельных логических строках, требуют выделения отступом. Если фрагмент состоит только из простых операторов, можно поместить их после двоеточия, в той же строке, через точку с запятой:

```
if a < b: a, b = b - a, a; print "a < b"
else: a, b = b, a - b; print "b <= a"
```

Конечно, этого следует избегать, т. к. программу становится тяжелее читать.

Структура этого раздела построена на достаточно условной классификации операторов языка Python и, как мы надеемся, позволит быстрее находить нужный оператор по оглавлению.

8.5.1. Операторы обработки данных

В эту группу собраны операторы, занимающиеся вычислениями (операторы-выражения и операторы присваивания), оператор вывода `print` и оператор удаления объектов и их частей `del`.

Оператор-выражение

Выражения являются полноценными операторами Python. При интерактивном использовании интерпретатора они печатают получаемый результат на стандартный вывод, а при появлении в программе просто теряют полученный результат. Например:

```
>>> 2+2
4
```

Этот же оператор в программе:

```
2+2
```

не будет печатать `4` при исполнении. Это не означает, что такие операторы бессмысленно использовать. При вычислении выражения могут измениться состояния участвующих в них объектов.

Например:

```
l = [2,3,1,4]
l.sort()
```

Второй оператор сортирует список `l` с помощью метода `sort()` и всегда возвращает значение `None` *(см. статью "Списки" в разд. 8.2.3)*.

Оператор присваивания

Операторы присваивания отличаются от операторов-выражений тем, что связывают вычисленное значение с одним или несколькими именами. Здесь мы не будем повторять все, что связано с выражениями *(см. разд. 8.3)* — они стоят в правой части присваиваний — а сосредоточимся на том, что стоит слева от "=".

Формально синтаксис присваивания можно записать так:

```
присваивание: (цели "=")+ выражения
цели:         цель ("," цель)* [","]
цель:         идентификатор | "(" цели ")" | "[" цели "]"
               | ссылка_на_атрибут | элемент | срез
```

Из этого описания видно, что в левой части от "=" могут быть целые структуры. Это, наверное, то, что отличает Python от многих других языков.

Листинг 8.28. Файл assigns.py. Присваивания

```
a = 1           # простейшие присваивания
b = 2
c = d = 3       # свяжем одно и то же значение с двумя именами
(a, b) = (b, a) # так можно поменять значения имен местами
a, c = c, a     # или так
print a, b, c, d
l = [1, 2, 3]   # связываем имя l со списком из трех элементов
k = l           # теперь и k связано с тем же списком
k[1] = -2       # меняем первый элемент списка (то есть 2)
print l         # печатаем список
k[1:1] = [1.1, 1.2]  # вклиним еще пару элементов в срез
print l
triangle = [(x1,y1),(x2,y2),(x3,y3)] = [(0,0),(1,0),(0,1)]
# здесь появляется сразу семь новых переменных
def square(x):  # определим функцию под именем square
   return x**2
sqr = square    # назовем функцию и другим именем
# a + b = "ab"  # а здесь была бы ошибка: слева не цель, закомментируем
```

```
a, b = "ab"      # присваиваем каждую букву отдельно
a, b = [1,2]     # главное, чтобы количество элементов слева и справа
                 # было одинаково
x = [0, 1]       # а в этом примере мы видим, что слева от '='
i = 0            # мы не может использовать присваиваемые переменные
i, x[i] = 1, 2   # здесь на втором месте уже не x[0], а x[1]
print "x =", x
```

Результаты:

```
3 1 2 3
[1, -2, 3]
[1, 1.1000000000000001, 1.2, -2, 3]
x = [0, 2]
```

Если слева стоит имя (идентификатор), с ним будет связано вычисленное значение. Имена в Python бывают *локальные* и *глобальные (см. разд. 8.7)*. Если имя было описано в операторе `global` *(см. статью "Оператор global" в разд. 8.5.4)*, оно будет принадлежать глобальному пространству имен. В противном случае имя будет локальным в данном блоке кода (внутри определения функции, класса или внутри модуля).

В Python 2.0 появилась разновидность присваивания, давно известная программистам C. Возможны следующие виды *расширенного присваивания*:

```
+=  -=  *=  /=  %=  **=  >>=  <<=  &=  ^=  |=
```

В целом `x += y` аналогично `x = x + (y)`, за исключением того, что `x` вычисляется один раз и, возможно, при исполнении операции используется некоторая оптимизация. (Заметьте также скобки вокруг `y`.)

Оператор *print*

Оператор `print` (от англ. *to print* — печатать) позволяет распечатывать объекты на стандартный вывод или в файл (или объект, похожий на файл). Он очень похож по синтаксису на одноименный оператор языка Basic. В версии 2.0 появилась расширенная форма оператора `print`. Если раньше он мог печатать только на стандартный вывод, то теперь для вывода можно назначить любой объект, который имеет метод `write()` (то есть умеет писать). В Python любой объект умеет представлять себя в виде строки. Это позволяет напечатать с помощью оператора `print` любой объект (хотя не любой объект будет при этом красиво выглядеть).

```
"print" [ expression ("," expression)* [","] ]
"print" ">>" expression [ ("," expression)+ [","] ]
```

Оператор вычисляет каждое выражение, переводит результирующие объекты в строки (как бы применяет `str()` — *см. статью "Функции преобразования ти-*

пов" в разд. 8.6.1) и выводит их. За исключением начала строки, перед выводом строки, соответствующей каждому объекту, выводится пробел, если:

- еще не было ни одной операции записи;
- последний записанный символ (перед данным оператором `print`) — символ новой строки `"\n"`;
- последняя запись делалась не оператором `print`.

Если оператор не заканчивается запятой, последним записывается символ `"\n"`. Объект, который может использоваться в операторе `print` для вывода, должен иметь метод `write()`.

Листинг 8.29. Всевозможные операторы `print`

```
print "*"*50
name = "pi"
pi = 3.14159
l = [3, 1, 4, 1, 5, 9]
print name, "==", pi, "цифры:", l
print `name`+" == " +`pi`+" цифры: "+`l`
print name+" == "+str(pi)+" цифры: "+str(l)
print "%(name)s == %(pi)f цифры: %(l)s" %vars()
print name, "==", pi,
print "цифры:", l
print """<html><head><title>многострочный вывод</title>
</head>
<body>
<p>Можно оформить так.</p>
<p>Пи == %(pi)f</p>
</body></html>""" % vars()
print "Теперь выведем пустую строку:"
print
print "Выводимые строки можно записать даже так. " \
      "Но это одна строка, " \
      "а не три."
print "*"*50
```

Результат выполнения:

```
**************************************************
pi == 3.14159 цифры: [3, 1, 4, 1, 5, 9]
'pi' == 3.1415899999999999 цифры: [3, 1, 4, 1, 5, 9]
```

```
pi == 3.14159 цифры: [3, 1, 4, 1, 5, 9]
pi == 3.141590 цифры: [3, 1, 4, 1, 5, 9]
pi == 3.14159 цифры: [3, 1, 4, 1, 5, 9]
<html><head><title>многострочный вывод</title>
</head>
<body>
<p>Можно оформить так.</p>
<p>Пи == 3.141590</p>
</body></html>
Теперь выведем пустую строку:

Выводимые строки можно записать даже так. Но это одна строка, а не три.
************************************************
```

Приведенный пример показывает, что подойти к решению задачи вывода данных можно по-разному. Какой именно способ выбрать — зависит от программиста. Следующий же пример демонстрирует запись на стандартный вывод разными способами:

```
import sys
sys.stdout.write("Java\n")
from sys import stdout
print >> stdout, "Python"
print "Basic"
```

Результат выполнения:

```
Java
Python
Basic
```

Стоит отметить, что `write()` не выводит конец строки `"\n"`, поэтому его приходится указывать явно. Это следует помнить при переходе от использования `print` к `write()` и обратно.

Оператор *del*

Оператор `del` служит для удаления каждого объекта из указанного в операторе списка (del — сокращение от англ. *to delete* — удалять).

```
del цели
```

`Цели` удаляются рекурсивно. Например, если в качестве `целей` указан список, сначала удаляются все элементы списка, после чего удаляется сам список. Если в качестве цели выступает имя, оно удаляется из соответствующего пространства имен. Реальное удаление связанного с этим именем объекта

происходит только если этот объект нигде больше не используется. Отдельными случаями являются удаления атрибутов, индексированных элементов или срезов. В этих случаях вызываются соответствующие методы затрагиваемых объектов.

Листинг 8.30. Использование оператора `del`

```
class a:
  def __del__(self):
     print "Меня удалили!"

s = { 1: 'apple', 2: 'orange', 3: 'lemon' }
l = [1, 2, 3, 4, 5]
o = a()
del s[3], l[1:3], o
print "s=", s
print "l=", l
a, b, c = 1, 2, 3
print "a=", vars().get('a', "?!")
del (a, [b, c])
print "a=", vars().get('a', "?!")
```

Результат выполнения:

```
Меня удалили!
s= {2: 'orange', 1: 'apple'}
l= [1, 4, 5]
a= 1
a= ?!
```

8.5.2. Управляющие операторы

Далее описаны операторы, влияющие на ход работы программы: оператор условия (выбора) `if`, операторы для организации циклов `while` и `for`, операторы прерывания и продолжения циклов `break` и `continue`, оператор возврата из подпрограммы `return` и пустой оператор `pass`.

Оператор *if*

Оператор условия `if` (в Python он же оператор выбора) — составной оператор, который последовательно проверяет истинность заданных в `if` и `elif` выражений и исполняет только один вложенный фрагмент программы, соответствующий первому истинному выражению. Если истинного

выражения не найдено, исполняется фрагмент, соответствующий части `else` (если она есть).

```
оператор_условия: "if" выражение ":" фрагмент
                  ("elif" выражение ":" фрагмент)*
                  ["else" ":" фрагмент]
```

Оператор условия легко прочесть если помнить, что `if` — "если", `else` — "иначе", а `elif` — сокращение от `else if`. Вложенный многострочный фрагмент должен иметь отступ, больший чем у оператора условия. Во вложенном фрагменте могут быть любые операторы, в том числе и другие операторы условия. Фрагмент может и однострочным. В этом случае он записывается в той же строке, что и `":"`. Об операторах *см. разд. 8.5.* О выражениях *см. разд. 8.3.*

Листинг 8.31. Применение условного оператора и оператора выбора

```
def abs(x):
  """Функция, возвращающая абс. значение числа"""
  # запись в одну строку
  if debug: print """abs(%(x)s)""" % vars()
  # оператор условия с двумя альтернативами
  if x >= 0:
    return x
  else:
    return -x

def sgn(x):
  """Функция, возвращающая знак числа"""
  if debug: print """sgn(%(x)s)""" % vars()
  # оператор выбора с тремя альтернативами
  if x > 0:
    return 1
  elif x < 0:
    return -1
  else:
    return 0

debug = 1
print abs(34)
print sgn(-23)
```

```
debug = 0
print abs(-34)
print sgn(23)
```

Результат выполнения:

```
abs(34)
34
sgn(-23)
-1
34
1
```

Оператор *while*

Оператор `while` служит для организации *циклов с предусловием*.

```
while_stmt:     "while" выражение ":" фрагмент1
              ["else" ":" фрагмент2]
```

Тело цикла — `фрагмент1` — выполняется, пока `выражение` дает истину. Если при очередной (хоть и первой) проверке условие не выполнено, выполняется `фрагмент2`. Внутри тела цикла `фрагмент1` может встретиться оператор `break`. В этом случае происходит выход из тела цикла без выполнения фрагмента `фрагмент2`. Оператор `continue`, выполняясь в теле цикла, передает управление обратно на проверку условия.

Замечание

Если не считать "необычной", но полезной добавки `else`, цикл `while` в Python аналогичен циклу `while` в C или `while-do` в Pascal. При начальном изучении программирования часть `else` лучше не использовать (не упоминать), чтобы не запутаться.

Листинг 8.32. Файл palindrom.py. Простой пример с циклом `while`

```
def palindrom(x):
  """Является ли строка палиндромом?"""
  s = x                # работаем над копией строки
  while len(s):        # пока строка не пуста:
    if s[0] != s[-1]:    # если первый символ не равен последнему:
      break                # строка не палиндром
    s = s[1:-1]          # берем строку без первого и последнего символов
  if len(s):           # возвращаем результат проверки
    return x + " не палиндром"
```

```
  else:
    return x + " палиндром"
print palindrom("арозаупаланалапуазора")
print palindrom("абракадабра")
```

Результат выполнения:

```
арозаупаланалапуазора палиндром
абракадабра не палиндром
```

Оператор *for*

В программировании очень часто возникают ситуации, когда необходимо выполнить некоторые действия над каждым элементом последовательности. В языке Python для этого есть несколько способов реализации, одним из которых является оператор цикла `for`.

```
оператор_for:  "for" сп_параметров "in" сп_выражений ":" фрагмент1
               ["else" ":" фрагмент2]
```

Здесь `сп_параметров` — последовательность параметров (аналогично целям в операторе присваивания *см. статью "Оператор присваивания" в разд. 8.5.1*), которым в ходе работы цикла присваиваются значения из последовательности `список_выражений`. Список выражений вычисляется один раз и должен дать последовательность, в противном случае возбуждается `TypeError: loop over non-sequence` (цикл не по последовательности). После того как для всех элементов последовательности тело цикла — `фрагмент1` — выполнилось, выполняется `фрагмент2` (если он есть). Если последовательность пустая, тело цикла не выполняется ни разу — сразу исполняется `фрагмент2` (если он есть).

Оператор `break` прерывает выполнение цикла и передает управление следующему за циклом оператору, даже не исполняя часть `else` *(см. статью "Оператор break" в разд. 8.5.2)*. Оператор `continue` заставляет перейти сразу к следующему элементу последовательности *(см. статью "Оператор continue" в разд. 8.5.2)*.

Для более привычной (по другим языкам программирования) работы с циклом `for`, можно использовать встроенные функции `range()` и `xrange()` *(см. статью "Функции обработки данных" в разд. 8.6.1)*.

Листинг 8.33. Сумма чисел от 1 до 100

```
S = 0
for i in range(1,101):
  S = S + i
print S
```

Результат выполнения:

```
5050
```

Внутри тела цикла `фрагмент1` можно присваивать другие значения параметрам цикла — на следующей итерации он получит новое значение. Некоторую осторожность следует соблюдать при изменении внутри `фрагмент1` последовательности, из которой получает значения параметра цикла. Реализация оператора `for` ориентируется по индексу элемента, и поэтому в случае изменения состава последовательности некоторые элементы могут быть пропущены или обработаны дважды. В любом случае лучше не изменять последовательность, используемую в теле `for`: это может привести к логическим ошибкам, которые будет трудно выявить. Если в ходе обработки последовательность все же должна измениться, для организации цикла лучше взять ее копию. Проще всего получить копию последовательности, выделив срез `[:]`:

```
>>> a = [1, 2, 3]
>>> for i in a[:]:
...   a[0:0] = [i-1]
...
>>> a
[2, 1, 0, 1, 2, 3]
```

Стоит заметить, что если бы в приведенном примере цикл осуществлялся не по копии `a`, он бы был бесконечным.

Следующий пример (листинг 8.34) показывает, что цикл может осуществляться и по объектам, эмулирующим последовательность:

Листинг 8.34. Файл fib1.py. Последовательность чисел Фибоначчи

```
class Fib:
  """Последовательность Фибоначчи"""
  def __init__(self, max):
    self.max = max
    self.n, self.a, self.b = 0, 0, 1
  def __getitem__(self, x):
    if self.n < self.max:
      a = self.a
      self.n, self.a, self.b = (self.n + 1, self.b, self.a + self.b)
      return a
    else:
      raise IndexError
```

```
for i in Fib(10):
  print i,
```

Результат выполнения:

```
0 1 1 2 3 5 8 13 21 34
```

То есть можно говорить, что (в данной реализации Python) цикл `for` начинает с 0-го элемента и действует до наступления исключения `IndexError` при попытке взять очередной элемент. В Python 2.2, с добавлением итераторов, ситуация будет проще *(см. разд. 8.11)*.

Оператор *break*

Оператор `break` прерывает выполнение цикла `while` или `for`, в котором непосредственно находится. Часть `else` (даже если она есть) при этом не выполняется, но если `break` передает управление вне тела фрагмента `try-finally`, часть `finally` все-таки выполняется. Счетчик цикла `for` сохраняет то значение, на котором цикл был прерван. Оператор `break` действует только в пределах блока кода *(см. разд. 8.7)*, т. е. он не может возвратить управление из определения функции.

Оператор `break` в сочетании с `while` можно использовать для организации цикла с пост-условием:

```
while 1:                # бесконечный цикл
  l = raw_input()         # читаем символы со стандартного ввода
  print ord(l)            # и выводим их коды
  if l == "/":            # завершаем цикл при вводе символа "/"
    break
```

Оператор *continue*

Оператор `continue` передает управление на начало цикла (в цикле `while` — на проверку условия, в цикле `for` берется следующее значение параметра цикла). До версии Python 2.1 оператор `continue` не может употребляться непосредственно в части `try` оператора `try-except`. Как и `break`, `continue` действует только в пределах блока кода.

```
print "Простые числа от 10 до 99"
for i in xrange(3, 100, 2):  # цикл от 3 до 99 с шагом 2
  if (i%3)*(i%5)*(i%7) == 0:   # если делится на 3, 5 или 7
    continue                     # то продолжаем искать
  print i,                     # печатаем простое число
```

Результат выполнения:

```
Простые числа
11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Оператор *return*

Оператор `return` служит для возврата из функции (или метода) и возвращения значения функции. Этот оператор можно задавать только внутри определения функции (метода) оператором `def`.

```
оператор_return:    "return" [список_выражений]
```

Если значение не указано, возвращается `None`.

```
def am(x, y):
  return x+y, x*y
print am(3, 4)
```

Результат выполнения:

```
(7, 12)
```

Оператор *pass*

Оператор `pass` ничего не делает. Его используют там, где по синтаксису положен оператор, но по логике программы не нужно выполнять никаких действий. Используется в качестве заглушки в определениях функций, классов и их методов, а также для игнорирования исключений в части `except` оператора `try-except`.

```
class A:
  pass

def nop(): pass

try:
  print 2/0
except:
  pass
```

8.5.3. Операторы обработки исключений

Во время исполнения программы могут возникать нештатные ситуации, называемые в Python исключениями. Для управления такими ситуациями служит оператор `try` в двух его разновидностях, программный вызов таких ситуаций можно осуществить с помощью оператора `raise`. Оператор `assert` помогает отлаживать программу, проверяя истинность записанного в нем утверждения.

Оператор *try*

Оператор `try` имеет две формы: `try-except-else` и `try-finally`:

```
try_except:  "try" ":" фрагмент
            ("except" [выражение ["," параметр]] ":" фрагмент)+
            ["else" ":" фрагмент]
try_finally: "try" ":" фрагмент
             "finally" ":" фрагмент
```

Оператор `try` предоставляет механизм для гарантированной обработки исключений (`try-except-else`) и исполнения определенных фрагментов кода при передаче управления вне оператора (`try-finally`).

Первая форма позволяет перехватывать выбранные исключительные ситуации, возникающие во `фрагменте` части `try`, и исполнять фрагмент кода соответствующего *обработчика исключений (exception handler)* одной из частей `except`.

Поиск подходящего обработчика исключений происходит по *идентификатору исключения* — объекту, который идентифицирует тип возникшего исключения. В ходе поиска части `except` проверяются по порядку, одна за другой. Поиск происходит до успешного совпадения идентификатора исключения со значением `выражения` в части `except`. Если `выражение` — кортеж, при поиске идентификатор исключения сравнивается со всеми элементами этого кортежа. Если в качестве выражения после `except` указан класс, то в качестве идентификатора подойдут и его классы-потомки. Часть `except` без выражения "ловит" и обрабатывает любые исключения. В поиске подходящего обработчика затрагиваются все операторы `try-except`, в которые вложен вызвавший исключение код, начиная с самого внутреннего и заканчивая самым внешним. В случае, когда исключение не было обработано ни одним из обработчиков, оно обрабатывается интерпретатором: выполнение программы прерывается и на `stderr` выводится информация об исключении.

Обработка исключения частью `except` начинается с присваивания `параметру` (если он задан) дополнительной информации по возбужденному исключению. После этого исполняется соответствующий `фрагмент` кода. Если в ходе вычисления `выражения`, присваивания `параметра` или исполнения `фрагмента except` возникают другие исключения, они обрабатываются внешними операторами `try-except-else`.

В обработчике исключения информация об исключении доступна посредством функции `sys.exc_info()` *(см. разд. 9.2.1)*.

Наконец, если в части `try` не произошло никаких исключений, выполняется часть `else` (если она есть) и управление передается следующему за `try-except-else` оператору. Впрочем, операторы `break` или `return` могут передать управление в соответствии со своим назначением. Оператор `continue` непосредственно внутри части `try` в версиях Python до 2.1 запрещен.

Листинг 8.35. Пример использования оператора `try-except`

```
def calc(x, y):
  """Вычисления, в которых может произойти исключение"""
  return x/(x-y)

try:
  S = 0
  for X in xrange(-1,2):
    for Y in xrange(-1,2):
      try:
        Z = calc(X,Y)  # спокойно производим вычисления
      except ZeroDivisionError, details:
        print "Ошибка при X=%(X)s, Y=%(Y)s : %(details)s" % vars()
        Z = 0   # если в них произошло деление на 0, делаем Z=0
      S = S+Z
  print "S =", S
except:
  print "Ошибка!"
```

Результат выполнения:

```
Ошибка при X=-1, Y=-1 : integer division or modulo
Ошибка при X=0, Y=0 : integer division or modulo
Ошибка при X=1, Y=1 : integer division or modulo
S = 2
```

В следующем фрагменте программы часть `except` ловит несколько разнородных исключений:

```
try:
  y = A[i]/x
except (IndexError, NameError), detail:
  print "Проблема с", detail
except ZeroDivisionError:
  print "Деление на нуль"
```

В некоторых случаях оператор `try-except-else` используют вместо оператора `if-elif-else`, чтобы не делать необходимых проверок по существованию и типу участвующих в обработке объектов.

```
d = { "a" : 1, "b" : 1 }
# вместо:
from operator import isNumberType
```

```
if d.has_key('a') and d.has_key('b') \
   and isNumberType(d['a']) and isNumberType(d['b']):
  c = d['a'] + d['b']
else:
  c = 0
# можно употребить:
try:
  c = d['a'] + d['b']
except:
  c = 0
```

В сложных случаях этот подход нужно применять с осторожностью, т. к. не до конца продуманная логика программы может привести к трудноуловимым ошибкам. При отладке рекомендуется либо временно убирать `try-except`, либо печатать информацию о возникающих исключениях в специальный файл.

Рассмотрим теперь вторую форму оператора — `try-finally`. Этот вид оператора позволяет гарантированно исполнить некоторый код в тех случаях, когда управление передается во внешний код фрагмента из кода части `try`. Исполнение фрагмента в `finally` происходит при возникновении исключений, применении операторов `return` или `break`, а также просто при нормальном завершении исполнения части `try`. Во время исполнения части `finally` информация о возбужденном исключении недоступна, а само исключение временно сохраняется и вновь возбуждается после завершения фрагмента `finally`. Операторы `try-except-else` и `try-finally` нельзя смешивать, но можно вкладывать друг в друга.

```
try:
  f = open("letter.tmp")      # открываем файл с письмом
  try:
    while 1:                    # цикл по строкам файла
      l = f.readline()
      if not l:                   # по окончанию файла
        break                       # выход из цикла
      print "> "+l                # печатаем знак цитаты
  finally:                      # в любом случае:
    f.close()                   # закрываем файл
    print "-- \nbye!"           # добавляем подпись
except IOError:
  print "<Ошибка ввода-вывода>"
```

Оператор *raise*

Исключения могут вызываться не только в результате выполнения встроенных функций, операций и операторов, но и явно. Для этой цели служит оператор `raise`.

```
оператор_raise: "raise" [выр1 ["," выр2 ["," выр3]]]
```

В табл. 8.21 показана зависимость идентификатора и параметра исключения от переданных оператору `raise` аргументов.

Таблица 8.21. *Аргументы оператора* `raise` *и возбуждаемое им исключение*

Тип `выр1`	Тип `выр2`	Идентификатор исключения	Параметр исключения
Строка `s`	Любой объект	Строка	`выр2`
Класс `cl`	`None`	`cl`	`cl()`
Класс `cl`	Кортеж `k`	`cl`	`apply(cl, k)`
Класс `cl`	Объект класса `cl` (или его потомков)	`cl`	`выр2`
Класс `cl`	Другой объект `obj1`	`cl`	`c(obj1)`
Объект `obj`	Ничего кроме `None`	`obj.__class__`	`obj`

Оператор `raise` без аргументов заново возбуждает (re-raise) обрабатываемое в `except` исключение. Кроме того, в операторе может быть задан третий аргумент — `выр3`. Этот аргумент должен быть трассировочным объектом *(см. статью "Трассировочные объекты" в разд. 8.2.10)*. С помощью него можно заново возбудить исключения, уже частично обработанные в части `except`, сохранив исходную трассировочную информацию.

```
def sqrt(x):
  """Квадратный корень"""
  if x < 0:
    # число должно быть неотрицательным: возбуждаем исключение
    raise TypeError, ("Отрицательный аргумент", x)
  else:
    return x**0.5

try:
  print sqrt(-1)
except TypeError, details:
  # Обрабатываем исключение, выводя сообщение об ошибке:
```

```
  print "Ошибка: %s: %s" % details.args
  # (здесь details -- объект класса TypeError)
```

Результат выполнения:

```
Ошибка: Отрицательный аргумент: -1
```

Утверждения

Одним из механизмов, позволяющих контролировать правильность выполнения программы, являются *утверждения*. Условие, записываемое в операторе утверждения `assert`, должно быть истинным. Если оно ложно, значит мы должны заподозрить сбой.

```
утверждение: "assert" expression ["," expression2]
```

где `expression` — выражение, условие для проверки; `expression2` — параметр исключения.

В варианте с одним выражением появление `assert` равносильно следующему:

```
if __debug__:
   if not expression: raise AssertionError
```

Когда задано `expression2`:

```
if __debug__:
   if not expression1: raise AssertionError, expression2
```

Примечание

Встроенная переменная `__debug__` обычно равна нулю, отключается заданием ключа `-O` при запуске Python.

Таким образом, оператор возбуждает исключение `AssertionError` в случае ложности `expression`.

```
lst = [1,2,-1]
try:
  s = 0
  assert len(lst), "Список не должен быть пустым."
  for l in lst:
    assert l > 0, "Числа должны быть положительными."
    s += l
except AssertionError, cause:
  print cause
```

Результат выполнения:

```
Числа должны быть положительными.
```

Как правило, assert должен использоваться именно для самоконтроля программы. В приведенном примере с помощью assert мы могли бы контролировать не просто корректность входных данных, а корректность самой программы, породившей эти данные. Таким образом, assert — одно из средств для отладки программ, которое можно включать и отключать. Для других целей лучше использовать механизм raise *(см. статью "Оператор raise" в разд. 8.5.3)*.

8.5.4. Декларативные операторы

Python имеет один *декларативный оператор* — оператор global.

Оператор *global*

Оператор global объявляет имена принадлежащими глобальному пространству имен и действует на весь блок кода, в котом он задан *(см. разд. 8.7)*. Без оператора global нельзя ничего присваивать глобальным именам, только читать. Документация по Python не рекомендует использовать глобальные имена до их указания в global, а также использовать их в качестве формальных параметров, параметров цикла for, в определении класса.

```
оператор_global:    "global" имя ("," имя)*
```

Оператор global относится строго к определенному блоку кода и не влияет на вызываемые или вызывающие данный блок другие блоки, т. к. global используется на этапе компиляции.

8.5.5. Операторы определения объектов

Операторы для определения функций и классов в языке Python не являются декларативными — они тоже исполняются, хоть и своеобразно. Ниже описаны операторы def и class для определения функций и классов соответственно.

Оператор *def*

Оператор def при своем исполнении определяет функцию, т. е., присваивает объекту функции *(см. статью "Функции" в разд. 8.2.5)* имя в локальном пространстве имен.

```
оп_def:        "def" имя_функции "(" [форм_арг_ты] ")" ":" фрагмент
форм_арг_ты:   (поз_арг ",")* (спец_арг | поз_арг [","])
спец_арг:      "*" имя [, "**" имя] | "**" имя
поз_арг:       параметр ["=" выражение]
подсписок:     параметр ("," параметр)* [","]
параметр:      имя | "(" подсписок ")"
```

Оператор def выполняет не только компиляцию кода фрагмента, но и вычисляет (и сохраняет) выражения, определяющие *значения аргументов по*

умолчанию. Аргументы без значений по умолчанию должны идти перед всеми другими аргументами. Смысл *специальных аргументов* объяснен в *статье "Вызовы функций и других объектов" в разд. 8.3.2*. На месте `параметра` может стоять и подсписок, что напоминает аналогичные возможности операторов присваиваний. (И действительно, при вызове функций происходят присваивания фактических аргументов формальным.)

Замечание

Тело определения функции не выполняется при ее определении, а только компилируется в кодовый объект. И наоборот: значения по умолчанию вычисляются только при определении функции и сохраняются в объекте-функции.

Ниже приведен пример определения функции, в котором представлены все разновидности аргументов:

```
def hairy(a, (b, c), d=4, *rest, **keywords):
  print "\n  a:", a, ", b:", b, ", c:", c, ", d:", d
  print "  rest:", rest
  print "  keywords:", keywords

print "1 hairy():", hairy(1, (2, 3), 40, 5, 6, x=-3, y=-2, z=-1)
print "2 hairy():", hairy(1, (2, 3), z=-1)
```

Результат выполнения:

```
1 hairy():
  a: 1 , b: 2 , c: 3 , d: 40
  rest: (5, 6)
  keywords: {'z': -1, 'x': -3, 'y': -2}
None
2 hairy():
  a: 1 , b: 2 , c: 3 , d: 4
  rest: ()
  keywords: {'z': -1}
None
```

Значение по умолчанию — одно и то же для всех вызовов. Следующий пример показывает, что изменяемые объекты (например, списки) лучше не использовать в этих целях.

```
def buggy_func(x=[]):
  x.append(1)
  return x
```

```
def buggy_func2(x=[]):
  return x

def better_func(x=None):
  if x == None:
    x = []
  x.append(1)
  return x

# Здесь с каждым вызовом значение по умолчанию меняется:
print buggy_func(), buggy_func(), buggy_func(), buggy_func([8])
# А здесь ошибка вообще неочевидна:
print "1-й вызов buggy_func2():", buggy_func2()
X = buggy_func2()
X[0:0] = ["surprise"]
print "2-й вызов buggy_func2():", buggy_func2()
# Теперь все нормально:
print better_func(), better_func(), better_func(), better_func([8])
```

Результат выполнения:

```
[1] [1, 1] [1, 1, 1] [8, 1]
1-й вызов buggy_func2(): []
2-й вызов buggy_func2(): ['surprise']
[1] [1] [1] [8, 1]
```

Определить объект-функцию можно не только с помощью оператора `def`, но и используя лямбда-выражение *(см. разд. 8.3.4)*. В последнем случае функция получается безымянной, а ее код может быть только выражением (тогда как в `def` можно использовать и другие операторы). Эти функции ничем не отличаются с точки зрения использования:

```
>>> def f1(x): return x+1
...
>>> f2 = lambda x: x+1
>>> print f1(5), f2(5)
6 6
>>> print f1, f2
<function f1 at 0x80db82c> <function <lambda> at 0x80d279c>
```

Оператор определения функции может встретиться и внутри определения функции. Только здесь стоит иметь в виду, что локальные имена будут недоступны в теле определяемой функции. Стандартный прием, позволяющий передать эти значения внутрь, — указать их в качестве значений по умолчанию. (Значения по умолчанию, как вы помните, вычисляются при определении функции). Следующий пример это демонстрирует:

Листинг 8.36. Функция для создания функций

```
def selector(*s):
  """Функция для создания функций выбора элементов"""
  def gi(x, lst=s):
    """Как бы шаблон функции выбора элементов"""
    return [ x[i] for i in lst ]
  return gi

s135 = selector(1, 3, 5)         # определяем селектор
print s135([1, 2, 3, 4, 5, 6])   # применяем селектор
print s135("abcdefg")            # для разных последовательностей
```

Результат выполнения:

```
[2, 4, 6]
['b', 'd', 'f']
```

В Python 2.2 будет возможность определять функции особого вида — генераторы. Об этом см. *разд. 8.11.*

Оператор *class*

Оператор определения класса `class` создает в локальном пространстве объект-класс с заданным именем.

```
оператор_class: "class" имя [наследован_от] ":" фрагмент
наследован_от:  "(" [список_выражений] ")"
```

Сначала вычисляется `список_выражений`, дающий список объектов-классов, которые станут базовыми для определяемого класса. Затем в новом фрейме исполнения *(см. разд. 8.7 и статью "Фрейм-объекты" в разд. 8.2.10)* исполняется `фрагмент`, для которого используется исходное глобальное и заводится новое локальное пространство имен. После выполнения `фрагмента` его локальные имена и вычисленный ранее список базовых классов используются для создания класса-объекта *(см. разд. 8.2.6)*. С этим объектом в исходном локальном пространстве связывается данное классу `имя`. И еще: все определенные непосредственно во `фрагменте` функции становятся методами класса.

Следующий пример иллюстрирует синтаксис описания класса и доступность методов и атрибутов:

Листинг 8.37. Файл class_ex1.py. Пример определения классов

```
class A:
  a = "A"
  def __str__(self):
    return self.a

class B:
  b = "B"
  def __len__(self):
    return 0

class AB(A, B):
  c = A.a + B.b
  d = B.b + A.a
  s = []
  def __init__(self, x):
    self.d = self.e = x
    self.s += [x]

a = AB(4)
b = AB(5)
# Атрибут объекта затеняет атрибут класса:
print "AB.d:", AB.d, "a.d:", a.d
# Атрибут класса -- общий для всех объектов:
print "a.s:", a.s, "b.s:", b.s
# Унаследованные методы:
print "str(a):", str(a)
print "len(a):", len(a)
```

Результат выполнения:

```
AB.d: BA a.d: 4
a.s: [4, 5] b.s: [4, 5]
str(a): A
len(a): 0
```

Классы можно применять (и в большинстве случаев так и делается) и без использования наследования (тем более множественного!). А те, кто скучает по структурам (C) или записям (Pascal), могут определять классы и без методов:

Листинг 8.38. Файл class_ex2.py. Класс как "запись" или "структура"

```
class struct:       # "чистый" класс
  pass

class record:       # а здесь применяются значения по умолчанию
  a = "default"
  b = 0

# Создаем "структуру"
s = struct()
r = record()
# Заполняем значениями:
s.name = "Иван Иванович Иванов"
s.phone = "7773332"
r.a = "my value"
# используем:
print "%(name)s -- %(phone)s" % s.__dict__
print r.a, "=", r.b
```

Результат выполнения:

```
Иван Иванович Иванов -- 7773332
my value = 0
```

Методы можно добавлять к классу даже после его определения простым присваиванием объекта-функции атрибуту класса. Еще более интересной является возможность пользоваться при определении класса внешними функциями, что позволяет создавать при конструировании классов более точные объектные модели, которые неуклюже описывать с помощью иерархии классов (так как таковая может потребовать искусственных абстрактных классов):

Листинг 8.39. Файл sig_poly.py. Пример конструирования классов из заготовленных ранее методов

```
def m_fly(self):
  return "Я летаю."
```

```
def m_swim(self):
  return "Я плыву."
def m_sing(self):
  return "Я пою."

class Everybody:
  def i_am(self):
    return "Я %s." % self.name
  def __getattr__(self, attname):
    def cant_do(self=None, action=attname):
      return "Я не умею %s." % action
    return cant_do

class Fish(Everybody):
  name = "Рыба"
  swim = m_swim

class Mermaid(Everybody):
  name = "Русалка"
  sing = m_sing
  swim = m_swim

class Bird(Everybody):
  name = "Птица"
  fly = m_fly
  sing = m_sing

class Man(Everybody):
  name = "Человек"
  sing = m_sing

f = Fish(); r = Mermaid(); b = Bird(); m = Man()
print f.i_am(), f.swim(), f.sing(), f.fly()
print r.i_am(), r.swim(), r.sing(), r.fly()
print b.i_am(), b.swim(), b.sing(), b.fly()
print m.i_am(), m.swim(), m.sing(), m.fly()
print "Людей научили плавать."
Man.swim = m_swim
print m.i_am(), m.swim(), m.sing(), m.fly()
```

Результат выполнения:

```
Я Рыба. Я плыву. Я не умею sing. Я не умею fly.
Я Русалка. Я плыву. Я пою. Я не умею fly.
Я Птица. Я не умею swim. Я пою. Я летаю.
Я Человек. Я не умею swim. Я пою. Я не умею fly.
Людей научили плавать.
Я Человек. Я плыву. Я пою. Я не умею fly.
```

8.5.6. Операторы исполнения

Операторы исполнения как бы запускают интерпретатор Python на отдельные блоки кода *(см. разд. 8.7)*. Оператор `import` (он употребляется в двух формах) помимо исполнения кода модуля может добавить имена, определенные в этом модуле, в пространство имен текущего исполняемого модуля. Оператор `exec` просто исполняет блок кода, данный ему в строке, открытом файле или в виде кодового объекта.

Оператор *import*

Оператор `import` позволяет импортировать модуль или (часть) его имен в Python-программу. Оператор существует в формах: `import` и `from-import`:

```
оп_import:        "import" модуль ["as" имя] ("," модуль ["as" имя] )*
оп_from_import:   "from" модуль "import" имя ["as" имя] ("," имя
["as" имя] )*
                | "from" модуль "import" "*"
модуль:           (имя ".")* имя
```

`оп_import` находит `модуль`, инициализирует его (если это необходимо) и связывает объект-модуль с именем `имя` в локальном пространстве имен. Так происходит со всеми заданными в операторе модулями.

`оп_from_import` тоже сначала находит `модуль` и инициализирует его, а затем помещает заданные в части `import` имена из этого модуля (меняя имя, если задан `as`). Звездочка `"*"` говорит оператору, что следует импортировать все имена, определенные в модуле (кроме тех, что начинаются с одиночного подчеркивания `"_"`).

Точный алгоритм поиска модуля зависит от реализации Python и компьютерной платформы. Обычно при этом выполняются следующие действия:

1. Поиск модуля в таблице ранее импортированных (и инициализированных) модулей `sys.modules`.
2. Поиск модуля среди встроенных модулей.

3. Поиск в соответствии со списком путей `sys.path`.
4. Если модуль не найден, возбуждается исключение `ImportError`.

(О поиске модулей, относящихся к пакетам и имеющих точки в имени см. разд. 8.8.)

После того как модуль найден (и соответствующий объект не находился в `sys.modules`), файл модуль.py компилируется (если соответствующий модуль.pyc или модуль.pyo имеют более раннюю дату создания, чем модуль.py) Откомпилированный байт-код записывается в файл модуль.pyc в том же каталоге, что и сам модуль (если у пользователя есть на это права доступа). Тем самым, в следующий раз модуль загрузится быстрее, не тратя времени на компиляцию.

После компиляции свежезагруженный модуль должен быть проинициализирован. Для этого он просто запускается на исполнение.

После того как получен объект-модуль, его имя записывается в текущем пространстве имен (если это оператор `import`) или (если это `from-import`) из модуля выбираются конкретные имена, и только они записываются в текущие локальные имена.

Процессы, связанные с импортом модулей, можно более тонко регулировать, задав свою функцию `__import__()`.

Чтобы импортировать новую версию модуля, используется не оператор `import`, а встроенная функция `reload()`, которая и перезагружает модуль. Кроме того, если для импорта была использована форма `from-import`, оператор `from-import` придется повторить после выполнения `reload()`.

О различных аспектах работы с модулями *см. разд. 8.2.7, 8.8, 8.7 и статью "Функции компиляции и исполнения" в разд. 8.6.1.*

Пример в следующих двух листингах прояснит вопросы использования оператора `import`:

Листинг 8.40. Файл mymodule1.py. Модуль `mymodule1`

```
"""Модуль для примера импорта"""
import string, re as _re
# _re не будет импортироваться по *
mysplit = string.split
_myresplit = _re.split # не будет импортироваться по *
# _myresplit не будет импортироваться по *
del string              # в составе этого модуля он не нужен
print "mymodule1 инициализирован"
```

Операторы `import` обычно ставят в начале программы или модуля. В следующей программе показаны всевозможные варианты импорта:

Листинг 8.41. Использование оператора `import`

```
import string as s, mymodule1, xml.dom.minidom
from string import split
from re import split as resplit
from mymodule1 import *      # повторный импорт: инициализации нет
from pprint import pprint
from xml.dom import minidom as mdom

print "dir(mymodule1):", pprint(dir(mymodule1))
print "1:", split("1,2,3", ","), resplit(",", "1,2,3"),
print "2:", s.split("1,2,3", ","), mymodule1.mysplit("1,2,3", ",")
# print _myresplit(",", "1,2,3")    # ошибка:
# имена, начинающиеся на подчеркивание не импортируются по *
print "3:", mymodule1._myresplit(",", "1,2,3")

d = mdom.Document()
d.appendChild(xml.dom.minidom.Element("mytag"))
print d.toxml()
```

Результат выполнения:

```
mymodule1 инициализирован
dir(mymodule1):['__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '_myresplit',
 '_re',
 'mysplit']
None
1: ['1', '2', '3'] ['1', '2', '3'] 2: ['1', '2', '3'] ['1', '2', '3']
3: ['1', '2', '3']
<mytag/>
```

Приведенный пример показывает, что `"as"` поможет избежать конфликта имен. В модуле `re` есть своя функция `split()`, поэтому мы обозначили ее `resplit()`. Кроме того, чтобы не засорять пространство имен модуля

mymodule1 мы удалили из него модуль string как только он стал не нужен. Реальные модули это делают не всегда, поэтому не стоит удивляться, найдя в модуле xml.dom.minidom модуль string — это тот же самый string, доступный в том числе через пространство имен модуля xml.dom.minidom.

Все имена модуля, которые нужно скрыть, лучше сделать с одиночным подчеркиванием в начале. Хотя такие имена все равно можно импортировать, программист сразу будет видеть, что имя предназначено для внутреннего использования.

Оператор *exec*

Оператор exec исполняет данный ему код.

```
оператор_exec:  "exec" выр1 ["in" выр2[","  выр3]]
```

где выр1 — код для исполнения; выр2 — глобальные переменные; выр3 — локальные переменные.

Выражение выр1 задает код для исполнения. Оно может быть строкой (перед исполнением она будет откомпилирована), открытым файловым объектом (откуда код будет считан, откомпилирован и исполнен) или готовым кодовым объектом (он будет просто исполнен). Словари выр2 и выр3 задают пространства имен для исполняемого кода. Если выр3 отсутствует, он считается равным выр2. Все изменения, которые происходят с именами в исполняемом exec коде, будут отражаться в этих словарях. Если часть in отсутствует, исполнение кода пройдет в текущем контексте. При исполнении кода могут произойти любые исключения. Специфичными для exec исключениями являются исключения класса SyntaxError, сигнализирующие о синтаксических ошибках в коде.

Кроме того, в текущей реализации Python можно заменить и встроенный модуль __builtin__, если в выр2 будет присутствовать имя __builtins__, указывающее на нужный модуль. Тем самым можно исполнять код в полностью контролируемых условиях (что обычно нужно для безопасности исполнения ненадежного кода).

Предупреждение

В текущей реализации код должен обязательно оканчиваться на "\n".

Листинг 8.42. Применение оператора exec

```
def mylen(x):
  """Функция сигнализирует о факте применения len()"""
  print "(меряли длину)"
  return len(x)
```

```
# Задаем имена:
g = {"__builtins__": {"len": mylen}}
l = {"a": 0}

exec """
a = a+1
b = [1, 2, 3, 5]
print "a=", a, "len(b)=", len(b)
""" in g, l
# доступ через словарь:
print "a=", l["a"], "len(b)=", len(l["b"])
# возможности кода ограничены: нет встроенных функций
exec """print "a=", a, "max(b)=", max(b)""" in g, l
```

Результат выполнения:

```
a= 1 len(b)= (меряли длину)
4
a= 1 len(b)= 4
a= 1 max(b)=
Traceback (most recent call last):
  File "<stdin>", line 17, in ?
  File "<string>", line 1, in ?
NameError: There is no variable named 'max'
```

8.6. Встроенные функции и исключения

В этом разделе будут рассмотрены все встроенные функции и исключения. Всего таких объектов в Python около сотни.

Формально встроенные функции и исключения принадлежат модулю `__builtin__`, но могут браться и из других источников, например, для обеспечения безопасности при исполнении ненадежного кода.

8.6.1. Встроенные функции

Встроенных функций в описываемой версии Python чуть более пятидесяти. Они служат в основном для порождения и преобразования встроенных типов данных.

Функции преобразования типов

Узнать тип объекта в Python можно при помощи функции `type()`. Например:

```
print type("string"), type(123), type(1+1j), type([1,2,3])
print type({'a':1,'b':2}), type(type(2))
```

Результат выполнения:

```
<type 'string'> <type 'int'> <type 'complex'> <type 'list'>
<type 'dictionary'> <type 'type'>
```

Приведение к строке. Любой встроенный тип данных в Python имеет два представления в виде строки: одно для удобства восприятия человеком, а другое является машинным представлением, по которому, в идеале, можно восстановить исходный объект. Конечно, это верно лишь для встроенных типов данных: чисел, строк, кортежей, списков, словарей при условии отсутствия в последних циклических ссылок. Другие объекты, например, функции, нельзя однозначно представить в виде строки и при их выводе оператором `print` приходится довольствоваться чем-то вроде:

```
<function f at 0x8113d14>
```

Между тем, даже приведение встроенных типов к строке (для восприятия человеком) может быть сделано несколькими способами. Например, число с плавающей точкой может содержать определенное количество цифр после точки, и даже сама строка может быть выведена с применением некоторых правил форматирования.

Любой объект, созданный пользователем, может позаботиться о своем представлении. Вот соответствующий пример:

Листинг 8.43. Объект, заботящийся о своем представлении

```
>>> class a:
...     def __repr__(x):
...         return "a()"
...     def __str__(x):
...         return "a"
...
>>> b = a()
>>> print b, `b`, str(b), repr(b), "%s" % b
a a() a a() a
```

В этом примере машинное представление объекта `b` класса `a` — строка `"a()"`, а "человеческое" — `"a"`. В списке для оператора `print` перечислено пять разных вариантов преобразования объекта `b` к строке. Первое, неявное, выполняется только оператором `print`. Во втором способе используется

машинное представление объекта. Третий и пятый способы основаны на явном преобразовании объекта к строке. Четвертый способ — получение "машинного" представления явным образом. Здесь стоит заметить, что внутри обратных апострофов (`) может стоять и выражение. Например:

```
>>> print `2+2`
```

Особенно ярко видны отличия str() от repr() в случае применения этих функций к собственно строке:

```
>>> print str("abc"), repr("abc")
abc 'abc'
```

Из этого примера видно, что "машинное" представление в идеале должно соответствовать синтаксически правильному описанию объекта в Python.

В табл. 8.22 приведены примеры различных встроенных типов данных в различных представлениях в виде строки.

Таблица 8.22. *Представление различных объектов в виде строки*

Тип	x	str(x)	repr(x)
Целое	12	12	12
Длинное целое	12L	12	12L
Строка	"abc"	abc	'abc'
Число с плавающей точкой	12.309	12.309	12.308999999999999
Комплексное число	2+4j	(2+4j)	(2+4j)
Список	[1,2,3]	[1, 2, 3]	[1, 2, 3]
Кортеж	(1,2,3)	(1, 2, 3)	(1, 2, 3)
Словарь	{'a': 1,'b': 2}	{'b': 2, 'a': 1}	{'b': 2, 'a': 1}

Модуль pprint поможет более красиво выводить списки, кортежи и словари *(см. разд. 9.2.8)*.

Другие преобразования. Как правило для преобразования одного типа к другому служит функция с названием целевого типа. В табл. 8.23 дан список операций для преобразования типов.

Таблица 8.23. *Функции для преобразования типов*

Тип	Функция для приведения к типу
Строка	str(), repr()
Целое	int()

Таблица 8.23 *(окончание)*

Тип	Функция для приведения к типу
Длинное целое	`long()`
Число с плавающей точкой	`float()`
Комплексное	`complex()`
Кортеж	`tuple()`
Список	`list()`
Словарь	

Примеры некоторых преобразований:

```
>>> list((1,2))
[1, 2]
>>> tuple([1,2])
(1, 2)
>>> tuple("abc")
('a', 'b', 'c')
>>> complex(123L)
(123+0j)
>>> int(-12.234)
-12
>>> int(12.234+4j)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to int; use e.g. int(abs(z))
```

Последний пример показывает, что не все преобразования возможны. Строки, кортежи и списки являются последовательностями, поэтому между ними всегда возможны преобразования. Числовые типы тоже могут быть преобразованы друг в друга, иногда с потерей информации (как происходит при преобразовании чисел с плавающей точкой в целые).

Привести типы к общему типу можно с помощью функции `coerce()`. Пример в листинге 8.44 показывает функцию `coerce()` во всевозможных ситуациях.

Листинг 8.44. Приведение к общему типу

```
def prtype(e):
  return map(lambda x: x.__name__, map(type,e))
```

```
numbers = [1, 1L, 1.0, 1j]
for i in numbers:
  for j in numbers:
    print prtype((i,j)), "--->", prtype(coerce(i,j))
```

Результат выполнения:

```
['int', 'int'] ---> ['int', 'int']
['int', 'long int'] ---> ['long int', 'long int']
['int', 'float'] ---> ['float', 'float']
['int', 'complex'] ---> ['complex', 'complex']
['long int', 'int'] ---> ['long int', 'long int']
['long int', 'long int'] ---> ['long int', 'long int']
['long int', 'float'] ---> ['float', 'float']
['long int', 'complex'] ---> ['complex', 'complex']
['float', 'int'] ---> ['float', 'float']
['float', 'long int'] ---> ['float', 'float']
['float', 'float'] ---> ['float', 'float']
['float', 'complex'] ---> ['complex', 'complex']
['complex', 'int'] ---> ['complex', 'complex']
['complex', 'long int'] ---> ['complex', 'complex']
['complex', 'float'] ---> ['complex', 'complex']
['complex', 'complex'] ---> ['complex', 'complex']
```

См. "Методы для организации преобразований типов" в разд. 8.4.3.

Числовые и символьные функции

Здесь мы рассмотрим различные встроенные функции, работающие с числами и символами. Сюда относятся как чисто числовые функции, так и некоторые преобразования.

Функция *abs()*. Функция `abs()` возвращает абсолютное значение числа. *(См. также разд. 9.4.1.)*

```
abs(x) -> y
```

Функция *divmod()*. Функция `divmod()` дает одновременно частное и остаток от деления своих аргументов.

```
divmod(x, y) -> (q, r)
```

где `q` — частное от деления `q` на `r`; `r` — остаток.

Функция *pow()*. Функция `pow()` предназначена для возведения в степень.

```
pow(x, y[, m]) -> z
```

где `x` — число, возводимое в степень; `y` — показатель степени; `m` — модуль; `z` — результат возведения `x` в степень `y` (по модулю `m`, если он присутствует).

Примечание

`pow(x, y, m)` аналогична `(x**y)%m`, но работает более эффективно. Например, чтобы узнать три последние цифры в записи числа `2**2343334`, достаточно задать `pow(2,2343334,1000)` и ответ будет получен за доли секунды, тогда как вычисление `pow(2L, 2343334)` заставит компьютер серьезно задуматься.

Функция *round()*. Функция `round()` служит для округления чисел до заданного знака после (или до) точки.

```
round(n[, z]) -> r
```

где `n` — округляемое число; `z` — до какого знака после точки округлять. По умолчанию 0. Отрицательные значения соответствуют номеру знака перед точкой.

```
n = 76543215.12345
for z in range(3, -4, -1):
  print "%2i : %14.5f" % (z, round(n, z))
```

Результат выполнения:

```
 3 : 76543215.12300
 2 : 76543215.12000
 1 : 76543215.10000
 0 : 76543215.00000
-1 : 76543220.00000
-2 : 76543200.00000
-3 : 76543000.00000
```

Предупреждение

В финансовых приложениях будьте очень внимательны, используя эту функцию или операцию форматирования `"%"`. Лучше заранее проверить правила округления положительных и отрицательных чисел, чем обнаружить ошибки в финансовых документах.

Функция *ord()*. Функция `ord()` возвращает код заданного ей символа (односимвольной строки). Для Unicode-строки эта функция возвращает код символа в Unicode.

```
for c in "012ABCabc":
  print ord(c),
print ord(u'\u1253')
```

Результат выполнения:

```
48 49 50 65 66 67 97 98 99
4691
```

Функция *chr()*. Функция `chr()` возвращает символ по заданному коду. Следующий пример выводит *таблицу ASCII*.

```
print "Таблица ASCII".center(7*8)
for k in range(32, 128):
  print "%3i: %1s" % (k, chr(k)),
  if (k+1) % 8 == 0:
    print
```

Результат выполнения:

```
                    Таблица ASCII
 32:    33: !  34: "  35: #  36: $  37: %  38: &  39: '
 40: (  41: )  42: *  43: +  44: ,  45: -  46: .  47: /
 48: 0  49: 1  50: 2  51: 3  52: 4  53: 5  54: 6  55: 7
 56: 8  57: 9  58: :  59: ;  60: <  61: =  62: >  63: ?
 64: @  65: A  66: B  67: C  68: D  69: E  70: F  71: G
 72: H  73: I  74: J  75: K  76: L  77: M  78: N  79: O
 80: P  81: Q  82: R  83: S  84: T  85: U  86: V  87: W
 88: X  89: Y  90: Z  91: [  92: \  93: ]  94: ^  95: _
 96: `  97: a  98: b  99: c 100: d 101: e 102: f 103: g
104: h 105: i 106: j 107: k 108: l 109: m 110: n 111: o
112: p 113: q 114: r 115: s 116: t 117: u 118: v 119: w
120: x 121: y 122: z 123: { 124: | 125: } 126: ~ 127:
```

Функция *len()*. Функция `len()` возвращает число элементов последовательности или словаря.

Функции *oct()* и *hex()*. Функции `oct()` и `hex()` возвращают строку с восьмеричным или шестнадцатеричным представлением заданного целого числа.

```
>>> print oct(127), hex(127)
0177 0x7f
```

Функция *cmp()*. Функция `cmp()` используется для сравнения двух значений.

```
cmp(x, y) -> integer
```

Результат сравнения `integer` отрицателен, если `x < y`, равен нулю при равенстве значений `x` и `y`, положителен, если `x > y`.

```
>>> print " "*26, map(cmp, [1, 2, 3, "abc"],
...                        [3, 2, 1, "bcd"])
                           [-1, 0, 1, -1]
>>> print cmp(None, Ellipsis)
-23
```

В Python 2.1 функция `cmp()` может принимать третий (необязательный) параметр, в котором можно явно указать вид сравнения: `"=="`, `"<"`, `">"`, `"<="`, `">="`, `"!="`. В этом случае возвращаемое значение может быть произвольным, подходящим по смыслу для данного сравнения.

Функция *unichr()*. Функция `unichr()` возвращает односимвольную Unicode-строку по заданному коду.

```
>>> print `unichr(3045)`
u'\u0BE5'
```

Функция *unicode()*. Функция `unicode()` создает Unicode-объект, соответствующий строке в заданной кодировке.

```
unicode(string [, encoding[, errors]]) -> obj
```

где `string` — строка в кодировке `encoding`; `encoding` — название кодировки (по умолчанию — UTF-8); `errors` — режим обработки ошибок (по умолчанию — `'strict'`).

Следующий пример выводит коды русских букв в Unicode:

```
print "Таблица Unicode (русские буквы)".center(18*4)
i = 0
for c in "АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"\
         "абвгдежзийклмнопрстуфхцчшщъыьэюя":
  u = unicode(c, 'koi8-r')
  print "%3i: %1s %s" % (ord(u), c, `u`),
  i += 1
  if i % 4 == 0:
    print
```

Результат выполнения:

```
                    Таблица Unicode (русские буквы)
1040: А u'\u0410' 1041: Б u'\u0411' 1042: В u'\u0412' 1043: Г u'\u0413'
1044: Д u'\u0414' 1045: Е u'\u0415' 1046: Ж u'\u0416' 1047: З u'\u0417'
1048: И u'\u0418' 1049: Й u'\u0419' 1050: К u'\u041A' 1051: Л u'\u041B'
1052: М u'\u041C' 1053: Н u'\u041D' 1054: О u'\u041E' 1055: П u'\u041F'
1056: Р u'\u0420' 1057: С u'\u0421' 1058: Т u'\u0422' 1059: У u'\u0423'
1060: Ф u'\u0424' 1061: Х u'\u0425' 1062: Ц u'\u0426' 1063: Ч u'\u0427'
1064: Ш u'\u0428' 1065: Щ u'\u0429' 1066: Ъ u'\u042A' 1067: Ы u'\u042B'
1068: Ь u'\u042C' 1069: Э u'\u042D' 1070: Ю u'\u042E' 1071: Я u'\u042F'
1072: а u'\u0430' 1073: б u'\u0431' 1074: в u'\u0432' 1075: г u'\u0433'
1076: д u'\u0434' 1077: е u'\u0435' 1078: ж u'\u0436' 1079: з u'\u0437'
1080: и u'\u0438' 1081: й u'\u0439' 1082: к u'\u043A' 1083: л u'\u043B'
```

```
1084: м u'\u043C' 1085: н u'\u043D' 1086: о u'\u043E' 1087: п u'\u043F'
1088: р u'\u0440' 1089: с u'\u0441' 1090: т u'\u0442' 1091: у u'\u0443'
1092: ф u'\u0444' 1093: х u'\u0445' 1094: ц u'\u0446' 1095: ч u'\u0447'
1096: ш u'\u0448' 1097: щ u'\u0449' 1098: ъ u'\u044A' 1099: ы u'\u044B'
1100: ь u'\u044C' 1101: э u'\u044D' 1102: ю u'\u044E' 1103: я u'\u044F'
```

Функции обработки данных

Здесь мы рассмотрим наиболее распространенные функции для обработки последовательных данных.

Функции *range()* и *xrange()*. Встроенные функции `range()` и `xrange()` служат для генерации последовательности из арифметической прогрессии чисел.

```
range([start,] stop[, step]) -> list
xrange([start,] stop[, step]) -> xrange_object
```

где `start` — с какого числа начинать последовательность (по умолчанию — 0); `stop` — до какого числа ее закончить; `step` — шаг арифметической прогрессии (по умолчанию — 1); `list` — список целых чисел, содержащий арифметическую прогрессию; `xrange_object` — специальный объект, умеющий генерировать числа последовательности по требованию (например, в операторе `for`).

```
>>> print range(5), range(1,5), range(1,5,2), xrange(1,5,2)
[0, 1, 2, 3, 4] [1, 2, 3, 4] [1, 3] xrange(1, 5, 2)
```

Стоит заметить, что в Python принято указывать диапазоны от `start` (включительно) до `stop` (исключительно), и функции `range()` и `xrange()` следуют этому правилу. Таким образом, последовательность индексов последовательности `s` задается просто с помощью `range(len(s))`. Например:

```
>>> lst = ['a','b','c']
>>> for i in xrange(len(lst)):
...   print i, lst[i]
...
0 a
1 b
2 c
```

Предупреждение

Функцию `xrange()` нежелательно использовать где-либо, кроме как в цикле `for`.

Функции *max()* и *min()*. Функции `max()` и `min()`, примененные к одному аргументу-последовательности вычисляют максимум (минимум) этой последо-

вательности. Если задано более одного аргумента, вычисляется максимум (минимум) среди аргументов.

- `max(sequence) -> value`
- `max(a1, a2, a3, ...) -> value`
- `min(sequence) -> value`
- `min(a1, a2, a3, ...) -> value`

```
>>> max(1,2,3,-1)
3
>>> min(1,2,3,-1)
-1
>>> max([1,2,3,4])
4
>>> min([1,2,3,4])
1
>>> max([1,3],[1,2],[2,0])
[2, 0]
>>> min([1,3],[1,2],[2,0])
[1, 2]
```

Функция *apply()*. Функция `apply()` (применить) позволяет вызывать объект на исполнение с заданными аргументами. Таким объектом может быть функция, класс и любой объект, имеющий метод `__call__()`. Функция обычно используется в случаях, когда готовый кортеж (или словарь) аргументов нужно подать на вход функции.

```
apply(object, args[, kwargs]) -> value
```

где `object` — вызываемый объект; `args` — кортеж аргументов; `kwargs` — словарь именованных аргументов.

Листинг 8.45. Различные варианты задания аргументов функции

```
def f(x,y,z):
   return (x,y,z)


xyz = (2,3,4)
xyz_dict = {'x':2, 'y':3, 'z':4}
print ( f(2,3,4) ==  f(2,3,z=4) ==  f(xyz[0],xyz[1],xyz[2])
        == apply(f,[2,3,4]) == apply(f,(2,3),{'z':4})
        == apply(f,xyz) == apply(f,(),xyz_dict) )
```

Результат выполнения:

```
1
```

Функция *map()*. Функция `map()`, пришедшая в Python, по-видимому, из LISP, — один из мощных инструментов функционального программирования, доступных в Python. Очень многие операции по обработке списков данных можно провести более эффективно при помощи этой функции. При этом объем кода тоже может существенно уменьшиться.

```
map (function, seq, ...) -> list
```

где `function` — функция; `seq, ...` — одна или несколько последовательностей, из которых берутся аргументы для функции; `list` — новый список, полученный применением функции `function()` к последовательностям.

Если `function == None`, то `list` будет состоять из кортежей соответствующих элементов последовательностей. Длина списка `list` будет равна длине самой длинной входной последовательности. Другие последовательности дополняются до нужной длины объектами `None`.

```
>>> from string import split
>>> lst = range(1,10)
>>> lst.reverse()
>>> print map(str,lst)
['9', '8', '7', '6', '5', '4', '3', '2', '1']
>>> print map(lambda x: x+1, lst)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>> ex = "1 2 3 4 5 6 7 8 9"
>>> ex1 = map(int, split(ex))
>>> print ex1
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> k = 1
>>> print map(max, lst, ex1)
[9, 8, 7, 6, 5, 6, 7, 8, 9]
>>> print map(lambda x,y,z=k: x*y+z, lst, ex1)
[10, 17, 22, 25, 26, 25, 22, 17, 10]
>>> print map(None, lst, ex1)
[(9, 1), (8, 2), (7, 3), (6, 4), (5, 5), (4, 6), (3, 7), (2, 8), (1, 9)]
>>> print map(None, range(5), range(7))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (None, 5), (None, 6)]
```

Сравните различные варианты реализации одного и того же:

Листинг 8.46. Увеличение каждого элемента списка на единицу разными средствами

```
lst = range(1000)  # исходный список
# С помощью оператора цикла:
newlst = []
for i in lst:
   newlst.append(i+1)
# С помощью map():
newlst = map(lambda x:x+1, lst)
# Python 2.0 появился еще один инструмент
# для работы со списками -- списковые включения:
newlst = [ x+1 for x in lst ]
```

Функция *filter()*. В соответствии со своим названием функция `filter()` фильтрует список, вызывая для каждого его элемента функцию с условием.

```
filter(function, sequence) -> list
```

где `function` — функция, возвращающая логическую истину или ложь. На место ее аргумента подставляются поочередно элементы последовательности; `sequence` — исходная последовательность; `list` — список всех элементов последовательности, для которых `function` возвращает истину.

```
lst = ['1', '', '2', '', '', '3']
print filter(len, lst)
#['1', '2', '3']
print filter(lambda x: int(x)>1, filter(len, lst))
#['1', '2']
allowed = "0123456789"
print filter(lambda x,a=allowed: x in a, "12 some text and 1 2 3")
```

Функция *reduce()*. Еще одна функция для обработки последовательностей — `reduce()` — позволяет реализовать цепочечное выполнение операций над элементами последовательности.

```
reduce(function, sequence[, initial]) -> value
```

где `function` — функция двух аргументов; `sequence` — последовательность, элементы которой обрабатываются; `initial` — значение по умолчанию для пустой последовательности (как бы приставляется к началу `sequence`); `value` — значение, получаемое в результате цепочки функций.

Наверное проще всего объяснить действие `reduce()`, написав эту функцию на Python:

Листинг 8.47. Эмуляция функции `reduce()`

```
def reduce(function, sequence, *initial):
  if len(initial) == 1:
    seq = [initial[0]] + sequence
  else:
    if len(sequence) == 0:
      raise TypeError, "reduce of empty sequence with no initial value"
    seq = sequence
  S = seq[0]
  for i in seq[1:]:
    S = function(S, i)
  return S
```

Это определение не совсем точная модель `reduce()`, но и она помогает понять, что происходит в данной функции. Проще всего проследить действие функции на примере со сложением. В математической записи такой `reduce()` обозначается большой сигмой, а на Python это будет выглядеть так:

```
>>> S = reduce(lambda x,y: x+y, [1, 2, 3, 4], 0)
>>> print S
10
```

Внутри же сложения происходят в следующем порядке:

```
S = ((1+2)+3)+4
```

Для умножения пример очень похож:

```
>>> P = reduce(lambda x,y: x*y, [1, 2, 3, 4], 1)
>>> print P
24
```

Здесь `P = ((1*2)*3)*4`.

Вы, наверное, заметили, что начальное значение `initial` в этом случае равно `1`, а не `0`. Это связано с тем, что в математике произведение нуля сомножителей равно `1`. (Просто `1` — ноль операции умножения.)

При программировании совсем необязательно, чтобы `initial` или `value` были числами. Ничто не мешает им быть растущими последовательностями или другими типами данных. Главное, чтобы функция `function()` понимала типы используемых данных.

Следует признать, что работа с `map()`, `filter()` и `reduce()` требует от программиста функционального стиля мышления (а не более распространенного процедурного). Обычно таким стилем мышления обладают программисты, применяющие LISP и ему подобные функциональные языки. Для остальных даже чтение приведенных выше конструкций может вызвать некоторую трудность. Тем не менее, именно функциональный подход (подкрепленный соответствующими языковыми конструкциями) приводит к более компактному и эффективному коду.

Функция *zip()*. Функция `zip()` позволяет объединить данные из нескольких последовательностей в одну последовательность кортежей.

```
zip(seq1 [, seq2 [...]]) ->
```

где `seq1`, `seq2`, `...` — последовательности; результат `[(seq1[0], seq2[0] ...), (...)]` — последовательность кортежей, в которой `i`-й кортеж состоит из `i`-х элементов исходных последовательностей. Результирующая последовательность имеет длину наиболее короткой из исходных последовательностей.

```
>>> print zip(range(6), "abcde")
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
>>> print map(None, range(6), "abcde")
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e'), (5, None)]
```

Стоит заметить, что в отличие от функции `map()`, `zip()` не добавляет `None`, чтобы удлинить слишком короткие последовательности, а просто обрубает результат по самой короткой последовательности.

С помощью `zip()` можно избежать использования индексов в циклах, где работа идет одновременно над несколькими последовательностями. Например:

```
>>> args = 1, 2, 100.2, "abc"
>>> fmts = "%5i", "%i", "%-4.2f", "%10s"
>>> for (fmt, arg) in zip(fmts, args):
...     print fmt % arg
...
    1
2
100.20
       abc
>>>
```

Функции определения свойств

Функции, которые мы здесь рассмотрим, позволяют получать различные характеристики и проверять свойства, имеющиеся у большей части объектов.

Функция *id()*. Функция `id()` возвращает гарантированно уникальный (среди одновременно существующих объектов) для каждого объекта номер. (В качестве номера используется адрес объекта в памяти).

```
id(object) -> integer
```

```
>>> s = 'abc'
>>> print map(id, [1, 1, 2,
...               'abc', 'ab'+'c', 'abc', s,
...               [1,2,3], [1,2,3]])
[134882168, 134882168, 134882120,
 135046648, 134996104, 135046648, 135046648,
 134996136, 135046696]
```

В этом примере видно, что одинаковые числа есть одинаковые объекты, тогда как текстуально одинаковые списки — разные объекты. Со строками сложнее. Здесь все зависит от того, получилась ли строка в результате операции или взята из константы: `'ab'+'c'` не тот же самый объект, что `'abc'`, тогда как `'abc'` и `s` — один объект. Очевидно, что если в блоке кода несколько раз встречается одна и та же строковая константа, Python хранит только один ее экземпляр. В остальных аспектах `'ab'+'c'` и `'abc'` совершенно неотличимы.

Функция *hash()*. `hash()` — встроенная в Python *хэш-функция*. Два одинаковых объекта дают одинаковое значение хэш-функции, но одинаковые значения хэш-функции не гарантируют одинаковости объектов, от которых они получены (хотя вероятность одинаковости объектов довольно высока).

```
hash(object) -> integer
```

```
>>> print map(hash, [1, 1, 2, 'abc', 'abc', 'ab'+'c', 'ab'])
[1, 1, 2, -1600925533, -1600925533, -1600925533, -1549758589]
```

Здесь мы видим, что одинаковые строки дают одинаковое значение функции `hash()`.

Функция *callable()*. Функция `callable()` проверяет, поддерживает ли объект вызов (как, например, функция или класс). Созданные пользователем классы поддерживают вызов, если имеют метод `__call__()`.

```
callable(object) -> Boolean
```

где `object` — некоторый объект; `Boolean` — 1, если объект можно вызвать на исполнение, и 0 в противном случае.

Листинг 8.48. Пример проверки аргументов на возможность вызова

```
>>> def mygetitem(f_or_l, a):
...   if callable(f_or_l):
```

```
...       return f_or_l(a)
...     else:
...       return f_or_l[a]
...
>>> def pseudo_list(i):
...     return i**2
...
>>> lst = [1,2,3,4,5]
>>> print mygetitem(lst, 3)
4
>>> print mygetitem(pseudo_list, 3)
9
```

Подобный прием используется для определения типа переданного функции объекта. В примере выше функция отрабатывает по-разному, в зависимости от того, функция ли первый аргумент или последовательность.

Функция *issubclass()*. Функция `issubclass()` определяет, унаследован ли класс `C` от класса `B`, возможно, через промежуточные классы.

```
issubclass(C, B) -> boolean
```

где `C и B` — классы; `boolean` — 1, если `C` унаследован от `B`, иначе — 0.

```
>>> print (issubclass(TypeError, StandardError),
...  issubclass(StandardError, TypeError),
...  issubclass(LookupError, StandardError),
...  issubclass(KeyError, LookupError),
...  issubclass(KeyError, StandardError))
(1, 0, 1, 1, 1)
```

Для краткости пример использует иерархию встроенных классов исключений.

Функция *isinstance()*. Функция `isinstance()` проверяет, принадлежит ли объект данному классу (или типу) или его потомкам.

```
isinstance(object, class-or-type) -> boolean
```

где `object` — проверяемый объект; `class-or-type` — класс или тип; `boolean` — 1 — принадлежит, 0 — нет.

```
>>> class A:
...   pass
...
>>> class B(A):
...   pass
```

```
...
>>> a = A()
>>> b = B()
>>> print isinstance(a,A), isinstance(a,B), isinstance(b,A), isin-
stance(b,B),
1 0 1 1
>>> import types
>>> print isinstance(1.2, types.IntType), isinstance(1.2,
types.FloatType)
0 1
```

Функция *type()*. Функция `type()` возвращает объект `type`, указывающий на тип объекта-аргумента.

```
type(object) -> type_object
```

где `object` — исследуемый объект; `type object` — объект, содержащий данные о типе `object`.

```
>>> print type('abc')
<type 'string'>
>>>
```

Функции для работы с системными объектами

Таблицы имен Python открыты для доступа из программ в виде словарей при помощи описанных ниже функций.

Функция *locals()*. Функция `locals()` возвращает словарь, представляющий пространство локальных имен. Изменения в данном словаре необязательно скажутся на локальных переменных. (То есть, словарь можно использовать только для чтения).

Функция *globals()*. Функция `globals()` возвращает словарь, представляющий глобальные имена. Этот словарь всегда относится к текущему модулю. Внутри функции или метода относится к модулю, в котором функция или метод определены. Изменения в словаре сразу сказываются на глобальном пространстве имен.

Функция *vars()*. Без аргументов функция `vars()` аналогична `locals()`. Если задан аргумент `obj`, вызов `vars(obj)` возвращает словарь данного объекта (аналогично `obj.__dict__`).

```
vars([obj]) -> dict
```

где `obj` — если задан, функция возвращает `obj.__dict__`; `dict` — словарь имен (данного объекта) или локальных имен.

Листинг 8.49. Применение `vars()` для заполнения шаблона значениями

```
name = "Иван Иванович Иванов"
address = "ivan@ivanov.ru"
subject = "Письмо"
year = 2002
my_name = "Петр Петрович Петров"
my_address = "petr@petrovich.ru"
print """To: "%(name)s" <%(address)s>
From: "%(my_name)s" <%(my_address)s>
Subject: %(subject)s

Привет, %(name)s!

С Новым, %(year)i, годом!

С уважением,
%(my_name)s

""" %vars()
```

Результат выполнения:

```
To: "Иван Иванович Иванов" <ivan@ivanov.ru>
From: "Петр Петрович Петров" <petr@petrovich.ru>
Subject: Письмо

Привет, Иван Иванович Иванов!

С Новым, 2002, годом!

С уважением,
Петр Петрович Петров
```

Как видно из примера, функция может быть очень полезна в связке с операцией форматирования. Она позволяет подставлять имена переменных в нужных местах.

Предупреждение

В случае, когда данные для письма берутся напрямую, например, из заполненной пользователем CGI-формы, приведенный вариант формирования электронного письма является небезопасным: ничто не мешает пользователю с недобрыми намерениями нарушить структуру письма и ввести собственные поля. Возьмите за правило всегда проверять вводимые пользователем значения с

точки зрения безопасности. Например, в нашем случае необходимо как минимум запретить использование символов возврата каретки во всех строках, вставляемых в шапку письма.

Функция *intern()*. Функция `intern()` добавляет к строку к специальной глобальной внутренней таблице интерпретатора. Это ускоряет доступ к строке. Функция возвращает саму строку или ранее помещенную строку с тем же содержимым.

```
intern(string) -> string
```

где `string` — строка.

Функция *dir()*. Функция `dir()` возвращает список имен атрибутов данного объекта (сюда не входят унаследованные атрибуты). При вызове без аргументов возвращает все видимые в данный момент локальные имена.

```
dir([obj]) -> strings
```

где `obj` — объект; `strings` — собственные атрибуты объекта `obj` в алфавитном порядке.

```
class A:
  attr1 = 1
  def mtd(self):
    lv = 3
    print "mtd:", dir()
  print "A:", dir()


class B(A):
  attr2 = 2
  print "B:", dir()

a = A()
b = B()
a.mtd()
b.mtd()
print "main:", dir()
```

Результат выполнения:

```
A: ['attr1', 'mtd']
B: ['attr2']
mtd: ['lv', 'self']
mtd: ['lv', 'self']
main: ['A', 'B', '__builtins__', '__doc__', '__name__', 'a', 'b']
```

Функции компиляции и исполнения

Функция *eval()*. Функция `eval()` вычисляет выражение, заданное ей в строке. Глобальное и локальное пространства имен также могут быть заданы (по умолчанию они совпадают с текущими).

```
eval (expression[, globals[, locals]]) -> result
```

где `expression` — выражение, дающее строку для вычисления или кодовый объект для исполнения; `globals` — словарь глобальных имен; `locals` — словарь локальных имен; `result` — результат вычисления.

```
>>> a = 7; b = -1
>>> d = {"a" : 3, "b" : 5}
>>> print eval("a*b", d)
15
>>> print eval("a*b", d, locals())
-7
```

Функция *execfile()*. Функция `execfile()` исполняет блок кода в файле с заданным именем. Если `globals` и `locals` не заданы, используются текущие пространства имен.

```
execfile (filename[, globals[, locals]])
```

где `filename` — строка с именем файла; `globals` — словарь глобальных имен; `locals` — словарь локальных имен.

Проанализируйте пример из следующих двух листингов:

Листинг 8.50. Файл exec-example

```
global b
a = 1
b = 2
c = c+1
```

Листинг 8.51. Программа, вызывающая exec-example

```
global b
b, c = 5, 2
execfile("exec-example")
print a, b, c
names = {"c" : 2}
a, b, c = -1, -2, -3
execfile("exec-example", names)
print a, b, c, names['a'], names['b'], names['c']
```

Результат выполнения:

```
1 2 3
-1 -2 -3 1 2 3
```

Функция *reload()*. Функция `reload()` перезагружает ранее импортированный модуль. Эта функция может пригодиться в случаях, когда файл с текстом модуля был изменен и требуется интегрировать эти изменения в программу без ее остановки. (Оператор `import` для этого не годится, т. к. он не загружает модуль снова).

```
reload (module) -> module
```

где `module` — модуль, который до этого был импортирован;

Перезагрузка модуля — процесс нетривиальный, и документация Python указывает несколько особенностей, которые следует иметь в виду.

- Модули расширений (написанные не на Python) могут неадекватно реагировать на перезагрузку. Перезагрузка же встроенных модулей не имеет практического смысла. Модули `sys`, `__main__`, `__builtin__` перезагружать не рекомендуется.
- Если модуль не содержит синтаксических ошибок, но его инициализация не была успешной (возбудила исключение), его имя не помещается в локальные имена. Частично инициализированный модуль помещается в словарь `sys.modules`. Для `reload()` его нужно снова импортировать, исправив ошибки, приводящие к сбою инициализации.
- При перезагрузке пространство глобальных имен модуля сохраняется. Имена, определяемые в новой версии модуля, будут переопределены, а те, что не затрагиваются инициализацией — останутся с прежними значениями. Это свойство можно использовать, проверяя наличие некоторого имени (например, обращаясь к нему внутри `try`) и только в случае его отсутствия определять имя.
- Если внутри модуля определяется некоторый класс, то при `reload()` методы существующих в программе объектов (созданных старой версией класса) не изменятся. Классы, производные от описываемых в модуле, так же не претерпят изменений.
- При перезагрузке модуля, часть имен которого была импортирована с помощью `from-import`, перезагрузки этих имен не произойдет. Для перезагрузки этих имен нужно повторить оператор `from-import`.

Листинг 8.52. Файл test.py. Модуль `test` (старый вариант)

```
global counter
try:
  counter
```

```
except:
  counter = 0

step = 1

class cnt:
  cnt_id = "oldclass"
  def show(self):
    global counter
    counter = counter+step
    return counter, step, self.cnt_id, "old"
```

Листинг 8.53. Файл test1.py. Модуль `test1` (новый вариант)

```
global counter
try:
  counter
except:
  counter = 0

step = 5

class cnt:
  cnt_id = "newclass"
  def show(self):
    global counter
    counter = counter+step
    return counter, step, self.cnt_id, "new"
```

Листинг 8.54. Испытание модуля

```
import os
import test             # старый вариант модуля
x = test.cnt()          # объект старого варианта модуля
print x.show()          # проверяем метод show()
# меняем старый вариант на новый
os.rename("test.py", "test0.py")
os.rename("test1.py", "test.py")
```

```
try:
  reload(test)          # перезагружаем модуль
  y = test.cnt()        # создаем новый объект
  print y.show()
  print x.show()
finally:
  # возвращаем исходные имена файлам
  os.rename("test.py", "test1.py")
  os.rename("test0.py", "test.py")
```

Результат выполнения:

```
(1, 1, 'oldclass', 'old')
(6, 5, 'newclass', 'new')
(11, 5, 'oldclass', 'old')
```

Проанализировав полученный результат, можно заметить следующее:

- переменная `counter` пережила перезагрузку модуля с прежним значением;
- атрибуты объекта `x`, созданного по определению класса в старом варианте модуля, не изменились;
- метод `x.show()` после перезагрузки модуля стал использовать новое значение `step`.

Этот пример еще раз демонстрирует продуманность реализации объектного программирования в Python: среда (в данном примере — модуль) может измениться, но объект остается целостным.

Функция *__import__()*. Функция `__import__()` встроенного модуля `__builtins__` вызывается операторами `import` и `from-import`. Вызывать эту функцию вместо оператора `import`, пожалуй, не имеет смысла, а вот ее переопределение позволяет держать под контролем все производимые в программе импорты модулей. Этот подход реализован в модуле `rexec` *(см. разд. 9.17)*, используемом для запуска небезопасного кода на Python в строгих рамках.

```
__import__ (name[, globals[, locals[, fromlist]]]) -> module
```

где `name` — имя модуля; `globals` — глобальные имена (не изменяются); `locals` — локальные имена (в текущей реализации не используются); `fromlist` — список имен, подлежащих импорту; `module` — объект.

Листинг 8.55. Замена функции `__import__` на свою функцию

```
def myimport(a,b,c,d,orig=__import__):
  """Замена для встроенного __import__"""
```

```
  print "Importing:", a, d  # только следим и затем
  return orig(a,b,c,d)      # вызываем исходный __import__
# Подменяем __import__:
__builtins__.__import__ = myimport
# Пробуем:
from string import join, split
import re
```

Результат выполнения (немного сокращен):

```
Importing: string ('join', 'split')
Importing: strop ('maketrans', 'lowercase', 'uppercase', 'whitespace')
Importing: re None
Importing: sre ('*',)
Importing: sre_compile None
Importing: _sre None
Importing: sre_constants ('*',)
Importing: sre_parse None
Importing: string None
Importing: sys None
Importing: sre_constants ('*',)
 ...
```

Из приведенного примера видно, что импорт модуля может вызвать еще несколько импортов других модулей или их частей, а также что `__import__` вызывается для каждого оператора `import`, даже если соответствующий модуль уже загружен.

Модуль `imp` (его описание не вошло в данную книгу) содержит все необходимое для конструирования своих реализаций `__import__()`.

Функция *compile()*. Функция `compile()` компилирует заданную ей строку в так называемый *кодовый объект*, пригодный для исполнения в операторе `exec` или в функции `eval()`. Возбуждает исключение `SyntaxError` в случае неуспеха компиляции.

```
compile (string, filename, kind) -> code_object
```

где `string` — строка, содержащая блок кода; `filename` — имя файла, из которого код был взят (или другая индикация источника кода, заданная в угловых скобках); `kind` — вид кода: `'exec'`, `'single'` или `'eval'`; `code_object` — кодовый объект.

Вид кода соответствует трем описанным в *разд. 8.1* случаям: `'exec'` задает файловый ввод программы, `'single'` — интерактивный (можно задать лишь один оператор, правда, с произвольным количеством вложенных) и

'eval' — ввод программы как перечисленных через запятую выражений. Скомпилированный код (то есть байт-код) при исполнении в eval или exec ведет себя в соответствии с заданным видом kind.

Листинг 8.56. Компиляция и исполнение кода

```
v = 100
a = compile("print 2*v\nprint v**2", "<string>", "exec")
b = compile("""\
for i in range(3):
  i
""", "<string>", "single")
c = compile("2*v, v**2", "<string>", "eval")
exec a
exec b
print eval(c)
```

Результат выполнения:

```
200
10000
0
1
2
(200, 10000)
```

Функции ввода/вывода

Далее рассмотрены функции языка Python для ввода/вывода.

Функции *raw_input()* и *input()*. Эти функции позволяют вводить значения со стандартного ввода:

- raw_input([prompt]) -> string
- input([prompt]) -> value

где prompt — строка приглашения ко вводу; string — введенная строка; value — введенное значение.

Функция raw_input() передает на стандартный вывод заданную строку приглашения (если она есть) и вводит строку, никак ее не интерпретируя. Функция input() аналогична eval(raw_input()), т. е. не только вводит стоку, но и интерпретирует ее.

Функция *open()*. Известная программистам на C функция `open()` для открытия файла:

```
open(filename[, mode[, buffering]]) -> file_object
```

где `filename` — имя файла; `mode` — режим открытия файла; `buffering` — способ буферизации; `file_object` — файловый объект.

Имя файла может быть полным с указанием абсолютного или относительного пути к файлу (считая с текущего каталога). Возможны следующие режимы открытия файлов:

- `"r"` — для чтения (этот режим действует по умолчанию);
- `"w"` — для записи;
- `"a"` — для добавления.

К букве, обозначающей режим, можно добавить модификатор:

- `"b"` — бинарный файл;
- `"t"` — текстовый файл;
- `"+"` — разрешить и запись, и чтение (режим обновления).

Модификаторы `"b"` и `"t"` имеют значение только на платформах, где текстовый файл требует преобразований символа конца строки, например Windows. В UNIX указание этих модификаторов применяется только для совместимости.

В табл. 8.24 показана реакция Python на различные ситуации при открытии файла.

***Таблица 8.24.** Действия, выполняемые при открытии файла в различных режимах*

Режим	Файл существует	Файл не существует
`"r"` (чтение)	Готов для чтения с начала	Исключение `IOError`
`"w"` (запись)	Файл усекается до пустого и готов для записи	Создается новый файл, готовый для записи
`"a"` (добавление)	Готов для записи в конец	Создается новый файл, готовый для записи

Дальнейшие операции по вводу/выводу реализуются как методы файлового объекта *(см. разд. 8.2.9)*.

Листинг 8.57. Открытие файлов в различных режимах

```
print "Записываем в файлы f1 и f2"
f1 = open("file1.txt", "wt")
```

```
f2 = open("file2.dat", "wb")
f1.write("1234\n")
f2.write("abcd")
f1.close()
f2.close()
print "Дописываем в файл f1"
f1 = open("file1.txt", "a")
f1.write("ABCD\n")
f1.close()
f2 = open("file2.dat", "r+")
print "f2:", `f2.read(4)`
f2.seek(0)
print "Пишем в начало файла f2"
f2.write("MN")
f2.close()

print "Читаем из файлов"
f1 = open("file1.txt", "rt")
f2 = open("file2.dat", "rb")
print "f1:", `f1.read()`
print "f2:", `f2.read(4)`
f1.close()
f2.close()
```

Результат выполнения:

```
Записываем в файлы f1 и f2
Дописываем в файл f1
f2: 'abcd'
Пишем в начало файла f2
Читаем из файлов
f1: '1234\012ABCD\012'
f2: 'MNcd'
```

Функции для работы с атрибутами

Функции `hasattr()`, `getattr()`, `setattr()` и `setattr()` дают необходимый интерфейс для работы с атрибутами через функции. Функция `hasattr()` проверяет наличие атрибута у объекта, `getattr()` — берет зна-

чение атрибута, `setattr()` — присваивает атрибуту значение, а `delattr()` удаляет атрибут.

- `hasattr (object, name) -> 0, 1`
- `getattr (object, name[, default_value]) -> obj`
- `setattr (object, name, value)`
- `delattr (object, name)`

где `object` — объект, у которого берется атрибут; `name` — строка с именем атрибута; `value` — присваиваемое атрибуту значение; `default_value` — значение на случай отсутствия атрибута; `obj` — объект, значение атрибута.

(См. статью "Ссылка на атрибут" в разд. 8.3.2.)

```
class A:
  pass


a = A()
a.att1 = 1
setattr(a, "att2", 2)
print "Атрибуты a:", dir(a)
print hasattr(a, "att3")
print getattr(a, "att2"), "==", a.att2
print getattr(a, "att3", 3)
del a.att1
delattr(a, "att2")
print "Атрибуты a:", dir(a)
```

Результат выполнения:

```
Атрибуты a: ['att1', 'att2']
0
2 == 2
3
Атрибуты a: []
```

Другие функции

Здесь будут рассмотрены функции для порождения объекта-среза и объекта-буфера.

Функция *slice()*. Функция `slice()` возвращает объект-срез (slice object) *(см. статью "Объекты-срезы" в разд. 8.2.10)*.

```
slice ([start,] stop[, step]) -> slice_object
```

где `start` — начало среза (по умолчанию `None`); `stop` — конец среза; `step` — шаг (по умолчанию `None`); `slice_object` — результирующий объект-срез.

Функция *buffer()*. Функция `buffer()` служит для создания объекта-буфера на основе другого объекта. Фактически буфер является массивом байтов.

```
buffer (object[, offset[, size]]) -> buffer_object
```

где `object` — строка, массив или объект-буфер; `offset` — смещение от начала буфера; `size` — размер буфера (по умолчанию — до конца `object`); `buffer_object` — результирующий объект-буфер.

Примечание

Функция настолько редко применяется, что во всех стандартных модулях она использована всего один раз и то только для того, чтобы получить тип объекта `buffer`!

8.6.2. Встроенные исключения

Исключение — ситуация, при которой интерпретатор не может продолжить исполнение кода и передает управление коду *обработчика исключений (exception handler)*. Исключение может быть возбуждено в случаях, нарушающих нормальный ход исполнения программы. Например, операция деления на ноль не может дать результата и возбуждает исключение `ZeroDivisionError`. Наряду со встроенными в программе могут быть определены и свои исключения *(см. статью "Оператор raise" в разд. 8.5.3)*.

Python имеет достаточно развитую иерархию встроенных классов-исключений. Они позволяют точно передавать суть ошибки в "охраняемом" `try-except` коде. *(Об этом операторе см. статью "Оператор try" в разд. 8.5.3.)* В последующих разделах мы рассмотрим практически все встроенные исключения и ситуации, в которых они возникают.

Следует заметить, что наследование исключений друг от друга — не чисто вопрос классификации. Оно позволяет более общим исключениям "ловить" ошибки, соответствующие их потомкам. Например:

```
>>> ['zero','one','two','three'][4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> try:
...   ['zero','one','two','three'][4]
... except LookupError:
...   print "Поймал!"
...
Поймал!
```

Это произошло потому, что `IndexError` наследуется от `LookupError`.

Практически все исключения (кроме `SystemExit`) — исключения-потомки `StandardError`, поэтому `StandardError` "поймает" любую ошибку (как и `try-except` без указания конкретного исключения).

Следующая программа выводит информацию об иерархии классов исключений:

Листинг 8.58. Файл exc_hier.py. Программа для вывода иерархии классов-исключений

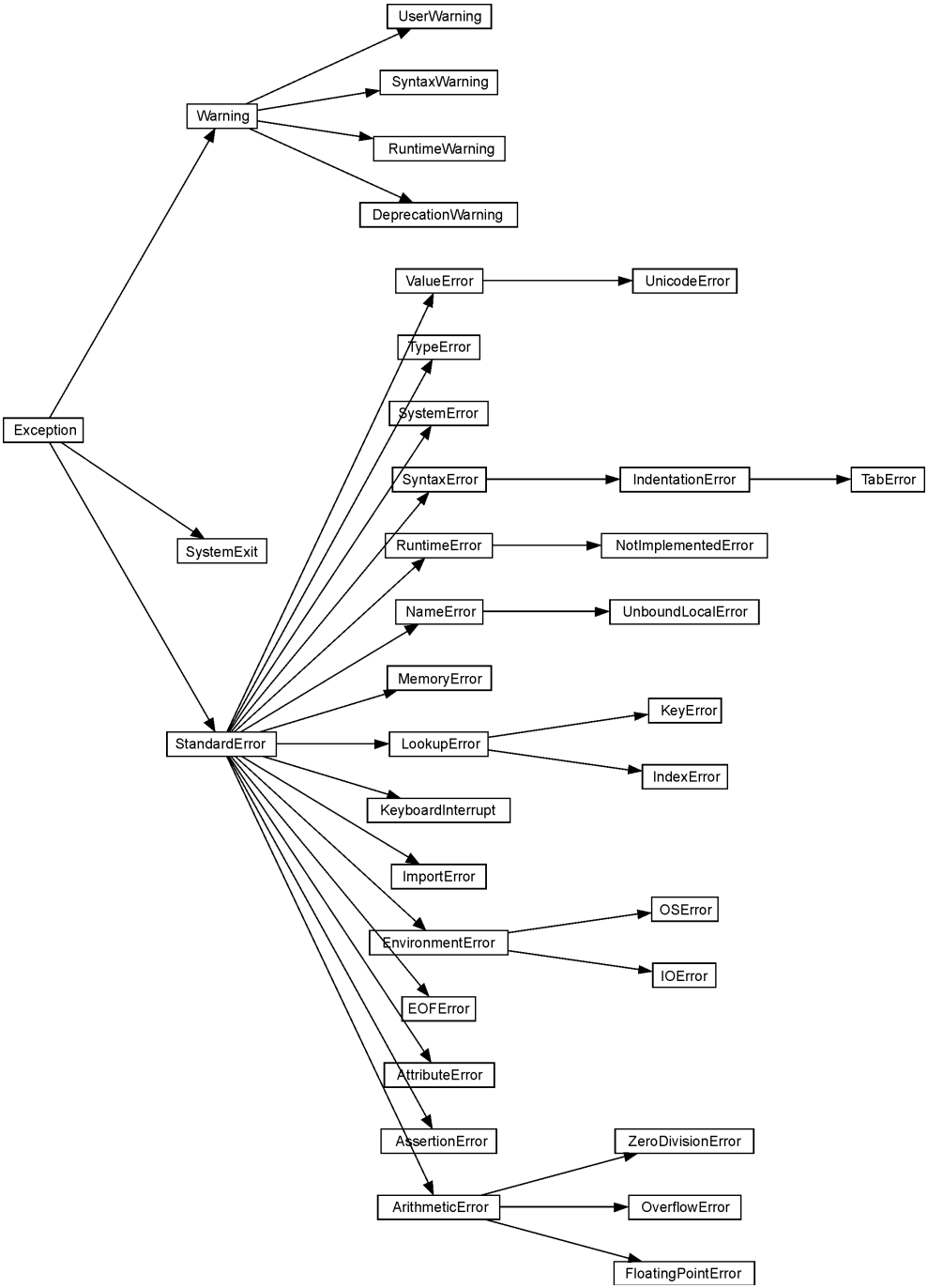
```
import types, string
d = {}  # инициализация словаря классов-родителей
for i in dir(__builtins__):  # для всех встроенных имен:
  # если тип класс и является потомком класса Exception:
  if types.ClassType == type(eval(i)) \
     and issubclass(eval(i), Exception):
    # выявляем базовые классы (у каждого класса-исключения
    # он только один)
    bases = map(lambda x: x.__name__, eval(i).__bases__)
    if bases:  # если базовые классы есть:
      # добавляем в словарь
      if not d.has_key(bases[0]):
        d[bases[0]] = []
      d[bases[0]].append(i)
# выводим в формате родитель : дети
for (parent, children) in d.items():
  print parent, ":", string.join(children, " ")
```

Результат работы этой программы будет зависеть от используемой версии Python. Более поздние версии Python имеют больше различных классов-исключений.

Базовые классы *Exception*, *SystemExit* и *StandardError*

Начиная с версии 1.5, Python перешел на исключения, основанные на классах. Это позволило не только более точно передавать суть ошибки через атрибуты объектов исключений, но и естественно расположить ошибки в дереве классов (рис. 8.3) в соответствии с общностью возникающих исключительных ситуаций.

Базовым классом для всех исключений является исключение `Exception`. От него происходят `SystemExit` (исключение возбуждается при выходе из программы) и `StandardError` — стандартная ошибка — базовый класс для всех других встроенных исключений.


UserWarning
SyntaxWarning
Warning
RuntimeWarning
DeprecationWarning
ValueError
UnicodeError
TypeError
SystemError
Exception
SyntaxError
IndentationError
TabError
SystemExit
RuntimeError
NotImplementedError
NameError
UnboundLocalError
MemoryError
KeyError
StandardError
LookupError
IndexError
KeyboardInterrupt
ImportError
OSError
EnvironmentError
IOError
EOFError
AttributeError
AssertionError
ZeroDivisionError
ArithmeticError
OverflowError
FloatingPointError

Далее мы рассмотрим встроенные исключения и причины их возбуждения подробнее.

ArithmeticError (базовый класс)

Исключение `ArithmeticError` объединяет исключения `OverflowError`, `ZeroDivisionError`, `FloatingPointError`, возбуждаемые интерпретатором при невозможности корректно выполнить арифметические операции.

AssertionError

Исключение `AssertionError` возбуждается при невыполнении записанного в операторе `assert` условия *(см. статью "Оператор assert" в разд. 8.5.3)*.

AttributeError

Исключение `AttributeError` возбуждается при безуспешной ссылке на атрибут или присваивании значения атрибуту. (Если объект вообще не поддерживает работы с атрибутами, возбуждается другое исключение — `TypeError`.)

```
>>> s = 123
>>> s.center
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'int' object has no attribute 'center'
# нет такого атрибута
>>> s.center = "c"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: attribute-less object (assign or del)
# объект не имеет атрибутов
>>> s = "123"
>>> s.cente = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object has read-only attributes
# все атрибуты объекта только для чтения
```

EnvironmentError (базовый класс)

Исключение `EnvironmentError` объединяет `IOError` и `OSError`.

```
>>> try:
...     open("nonexistent", "r")    # открываем несуществующий файл
```

```
... except EnvironmentError, details:
...   print details.args, details.filename
...
(2, 'No such file or directory') nonexistent
```

FloatingPointError

Исключение `FloatingPointError` возбуждается при ошибке в операции с плавающей точкой. (Работает только если Python был скомпилирован с опцией `WANT_SIGFPE_HANDLER`.)

ImportError

Исключение `ImportError` возбуждается в случае невозможности импорта модуля или некоторых имен из модуля.

```
>>> from string import compile
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: cannot import name compile
```

IndentationError

Исключение `IndentationError` возбуждается при наличии неправильных отступов в исполняемом коде. Подкласс `SyntaxError`. Например:

```
>>> for ii in 1,2,3:
...    print ii
...  print ii
  File "<stdin>", line 3
    print ii
          ^

IndentationError: unindent does not match any outer indentation level
```

IndexError

Исключение `IndexError` возбуждается в случае, когда элемента последовательности с таким индексом не существует. Класс `IndexError` наследуется от `LookupError`. Данная ошибка не возникает при выделении срезов: они автоматически приводятся к нужному диапазону. В случае же, когда индекс не является целым, возбуждается исключение `TypeError`.

```
>>> ['zero','one','two','three'][4]
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
IndexError: list index out of range
# индекс за пределами допустимого диапазона
```

IOError

Исключение `IOError` применяется для обработки ситуаций, связанных со вводом/выводом. Класс `IOError` наследуется от класса `EnvironmentError`.

```
file = open("NOFILE", "r")
```

Результат выполнения:

```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'NOFILE'
# Нет такого файла 'NOFILE'
```

KeyError

Исключение `KeyError` возбуждается в случае отсутствия ключа в словаре (соответствии).

```
>>> vars()['3']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 3
```

KeyboardInterrupt

Исключение `KeyboardInterrupt` возбуждается в ответ на нажатие пользователем комбинации клавиш, прерывающих исполнение программы (обычно <Ctrl>+<C> или <Del>). Это исключение возбуждается и при ожидании ввода в функциях `input()` и `raw_input()`.

```
>>> for i in xrange(100000):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyboardInterrupt
>>>
```

Об обработке других системных событий *см. разд. 9.7.10.*

LookupError (базовый класс)

Исключение `LookupError` объединяет исключения, связанные с доступом к элементам последовательностей и словарей, является базовым классом для `IndexError` и `KeyError`.

MemoryError

Исключение `MemoryError` возбуждается при невозможности выполнить операцию ввиду отсутствия свободной памяти в системе. В некоторых случаях ситуацию можно выправить, удалив лишние объекты (или отказавшись от создания крупного объекта).

```
>>> a=range(10000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
MemoryError
>>>
```

NameError

Исключение `NameError` происходит при ненахождении некоторого имени.

```
>>> b=a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: There is no variable named 'a'
# нет переменной с именем 'a'
```

NotImplementedError

Исключение `NotImplementedError` является производным от `RuntimeError`. По правилам хорошего тона следует поднимать это исключение в абстрактных методах базовых классов, требуя их переопределения в классах-наследниках.

```
class A:
  def show_something(me, x):
    raise NotImplementedError, "здесь заглушка"

class A1(A):
  def show_something(me, x):
    print x*"*"

A1().show_something(10)
A().show_something(10)
```

Результат выполнения:

```
**********
Traceback (most recent call last):
  File "<stdin>", line 9, in ?
  File "<stdin>", line 3, in show_something
NotImplementedError: здесь заглушка
```

OSError

Исключение `OSError` используется в основном функциями модуля `os` для сигнализации об исключительных ситуациях при работе с функциями операционной системы. Происходит от класса `EnvironmentError`.

```
import os
os.stat("NOFILE")
```

Результат выполнения:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
OSError: [Errno 2] No such file or directory: 'NOFILE'
# Нет такого файла 'NOFILE'
```

OverflowError

Исключение `OverflowError` возбуждается, если результат арифметической операции слишком велик для представления в имеющемся типе данных. Следующий пример демонстрирует невозможность сменить знак у наименьшего числа целого типа. В то же время операция левого побитового сдвига ошибки не вызывает.

```
>>> minint = (-2)**31
>>> minint
-2147483648
>>> -(minint+1)
2147483647
>>> -minint
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
OverflowError: integer negation
>>> # однако:
>>> 2<<31
0
```

RuntimeError

Исключение `RuntimeError` происходит при возникновении ошибки периода исполнения, не попадающей ни в какую другую категорию. До сих пор используется лишь в некоторых стандартных модулях.

SyntaxError

Самое популярное "исключение" у начинающих пользователей, исключение `SyntaxError`, знаменует собой синтаксическую ошибку в разбираемом коде. Разумеется, такое исключение не может появиться в главном модуле, т. к. до его исполнения дело просто не дойдет, но вполне реально появление такого исключения из-за ошибок в модулях и других блоках кода, компилирующихся в байт-код во время исполнения основной программы.

```
>>> eval("2+)
  File "<stdin>", line 1
    eval("2+)
           ^
SyntaxError: invalid token
>>> # Синтаксическая ошибка: неверная лексема
>>> try:
...   eval("2+")
... except SyntaxError, x:
...   print x.args
...
('unexpected EOF while parsing', (None, 1, 2, '2+'))
>>> # Синтаксическая ошибка: неожиданный конец программы
>>> # (приводятся координаты ошибки)
```

SystemError

Исключение `SystemError` возникает при внутренней ошибке в интерпретаторе Python. В случае ее возникновения разработчики Python просят выслать им сообщение об ошибке, номер версии Python (`sys.version`), условия возникновения ошибки (например, исходный код, вызывающий ошибку).

Автору данной книги сложно привести пример `SystemError`!

TabError

Исключение `TabError` является подклассом `SyntaxError` и происходит в случае неправильного применения смеси пробелов и табуляций для обозначения отступов.

TypeError

Исключение `TypeError` возбуждается в случае, если операция или функция применяется к объекту неподходящего типа.

```
>>> print "a" - "b"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for -
# Неверный тип операнда для -
```

UnboundLocalError

Исключение `UnboundLocalError` возбуждается в случае, когда идет обращение к локальной переменной, которой еще не было присвоено значение. Является подклассом `NameError`. Следующий пример демонстрирует типичный повод для данного исключения.

```
>>> def bad_function(x):
...   if x == 0:
...     v = 1
...   else:
...     v = v+x
...   return v
...
>>> print bad_function(0)
1
>>> print bad_function(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in bad_function
UnboundLocalError: Local variable 'v' referenced before assignment
# переменная 'v' использована до присваивания
```

Здесь присваивание происходит в альтернативной ветке оператора `if` и потому переменная оказывается непроинициализированной.

UnicodeError

Исключение `UnicodeError` является подклассом `ValueError` и возбуждается в случае возникновения проблем с кодированием или декодированием в Unicode.

Следующий пример:

```
>>> print unichr(1111)

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
# ошибка кодирования: код не в пределах 0-127
```

ValueError

Исключение `ValueError` происходит в случае, когда операции или функции передается аргумент нужного типа, но неподходящего значения.

```
>>> 2**-2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: integer to a negative power
# целое число нельзя возводить в отрицательную степень
```

WindowsError

Исключение `WindowsError` является подклассом `OSError` для платформы Windows. Возбуждается в случае ошибки, специфичной для этой платформы.

ZeroDivisionError

Исключение `ZeroDivisionError` является подклассом `ArithmeticError` и служит для сигнализации о делении на ноль в операциях деления и взятия остатка от деления.

8.7. Пространства имен

Для начала определим понятие *блока кода (code block)* и *фрейма исполнения (execution frame)*. Блок кода — кусок программы на Python, который может быть исполнен как единое целое (возможно, с исполнением кода из других блоков, текстуально содержащихся или не содержащихся в данном блоке). Блоками кода являются:

- модуль;
- тело определения функции;
- определение класса;
- команда, заданная в интерактивном режиме;
- сценарий, который может исполнить интерпретатор Python;

- команда, заданная в командной строке ключом `-c` при вызове интерпретатора;
- файл, выполняемый встроенной функцией `execfile()`;
- вычисляемый аргумент функции `eval()`;
- аргумент оператора `exec`;
- выражение, интерпретируемое функцией `input()`.

Блок кода выполняется в среде фрейма исполнения. Фрейм содержит информацию, необходимую для отладки, о передаче управления по завершении исполнения данного блока кода, о глобальных и локальных именах.

Пространство имен устанавливает соответствие между идентификаторами (именами) и объектами. Оно аналогично словарю, ключами для которого являются имена, а значениями — соответствующие этим именам объекты.

В отличие от некоторых других языков программирования, язык Python имеет всего три неявных пространства имен (куда входят все появляющиеся в программе имена: от имен классов и модулей до рядовых переменных) и простые правила при работе с ними. Естественно, что одно и то же пространство имен может использоваться несколькими блоками кода.

Говорят, что имя закрепляется (binding) за пространством имен, когда к пространству добавляют новое имя. Перезакрепляется (rebinding), если имя закрепляется за другим объектом. Открепляется (unbinding), если имя удаляется из пространства имен. В других разделах данной книги мы пользуемся более привычными терминами: присваивание, переприсваивание и удаление переменной, хотя приведенные выше термины более точно отражают специфику языка Python по работе с именами. Если в языке C мы привязываем имя к конкретной области памяти и потом можем сколько угодно менять ее содержимое, то в Python все объекты от рождения безымянны и могут получить сколько угодно имен. Объект существует до тех пор, пока хотя бы один другой объект на него ссылается. Различные случаи закрепления имен приведены в табл. 8.25.

Таблица 8.25. *Случаи закрепления имен*

В каком случае	Что закрепляется	Примечание
Для формальных аргументов функций	Имя аргумента	
При определении класса или функции	Имя класса или функции	
В операторе `import`	Имя импортируемого модуля	Имя модуля дает доступ к пространству имен модуля

Таблица 8.25 *(окончание)*

В каком случае	Что закрепляется	Примечание
В операторе `from-import`	Все или часть имен из пространства имен модуля закрепляются в данном блоке	При этом имена могут изменяться, если используется `as`
В операторе присваивания	Цели, указанные в левой части присваивания	
В цикле `for`	Параметр цикла	
Во втором аргументе части `except` оператора `try-except`	Параметр исключения	

Локальное пространство имен служит для определения и поиска имен в данном блоке. *Глобальное пространство имен* распространяется на имена, указанные в операторе `global`, а также имена, которые не закрепляются нигде в данном блоке. (Перед исполнением блока кода интерпретатор просматривает блок определяя, закрепляется ли данное имя где-либо в нем). Более формально:

```
если имя упомянуто в операторе global данного блока:
  имя глобальное
иначе если имя не закреплено нигде в данном блоке:
  если имя находится в __builtins__:
    имя встроенное
  иначе:
    имя глобальное
иначе:
  имя локальное
```

Поиск значения имени в пространствах имен происходит по следующей схеме:

```
если имя находится в пространстве локальных имен:
  берется значение из пространства локальных имен
иначе если имя находится среди глобальных:
  берется значение из пространства глобальных имен
иначе если имя находится в словаре __builtins__:
  берется значение из __builtins__ (обычно это имя имеет модуль
__builtin__, поэтому берется глобальное значение из этого модуля)
иначе:
  возбуждается исключение NameError
```

В некоторых случаях пространства имен можно задать явно. Табл. 8.26 поможет разобраться, что в каком случае является глобальным и локальным пространством имен. (Далее вместо словосочетания "пространство имен" употребляется слово "имена").

Таблица 8.26. *Пространства имен различных видов блоков кода*

Блок кода	Глобальные имена	Локальные имена
Модуль	Имена модуля	Совпадают с глобальными
Сценарий (файл или команда)	Имена верхнего уровня	Совпадают с глобальными
Интерактивная команда	Имена верхнего уровня	Совпадают с глобальными
Определение класса	Что и глобальные у объемлющего блока	Новые
Определение функции	Что и глобальные у объемлющего блока	Новые
Определение метода	Что и глобальные у объемлющего блока	Новые
Аргумент оператора `exec`	Может быть задан (по умолчанию что и у объемлющего блока)	Может быть задан (по умолчанию что и у объемлющего блока)
Аргумент функции `eval()`	Может быть задан (по умолчанию что и у объемлющего блока)	Может быть задан (по умолчанию что и у объемлющего блока)
Файл в `execfile()`	Может быть задан (по умолчанию что и у объемлющего блока)	Может быть задан (по умолчанию что и у объемлющего блока)
Строка, введенная по `input()`	Что и глобальные вызывающего блока	Что и локальные у вызывающего блока

Примечание

Если `eval()`, `exec` или `execfile()` задан только один параметр пространства имен, глобальные и локальные имена берутся из него.

Этот раздел описывает пространства имен Pyhton 2.0 и более ранних версий. Версия 2.1 (в качестве опции) и проектируемая версия 2.2 (в обязательном порядке) будут иметь несколько иные правила. При этом переходе большинство кода, написанного на Python, не пострадает.

Начиная с версии 2.1, Python использует более естественные правила для пространств имен. Для включения механизма *вложенных областей действия имен (nested scopes)* в Python 2.1 необходимо указать:

```
from __future__ import nested_scopes
```

что можно прочитать "как: из будущего импортировать вложенные области действия (имен)".

Основное изменение состоит в том, что теперь во вложенных блоках кода можно использовать имена из внешних блоков, например:

```
#!/usr/bin/python2.1
def f(x):
  def g(y):
    print x, y
  g(x+1)
f(1)
```

Будет выполнен в Python 2.1 как:

```
./try.py:6: SyntaxWarning: local name 'x' in 'f' shadows use of 'x' as
global in nested scope 'g'
  def f(x):
Traceback (most recent call last):
  File "./try.py", line 10, in ?
    f(1)
  File "./try.py", line 9, in f
    g(x+1)
  File "./try.py", line 8, in g
    print x,y
NameError: global name 'x' is not defined
```

То есть, во-первых, Python выдаст предупреждение (warning), а, во-вторых, произойдет ошибка из-за того, что переменной `x` "не видно" внутри функции `g`. Тогда как в следующем примере все будет нормально:

```
#!/usr/bin/python2.1
from __future__ import nested_scopes
def f(x):
  def g(y):
    print x, y
  g(x+1)
f(1)
```

Результат выполнения:

```
1 2
```

Из новых правил областей действия имен есть исключение: имена, закрепленные в блоке определения класса, не будут видны во вложенных в него блоках. Это сделано для того, чтобы не нарушать принятых в Python правил определения методов класса.

Кроме того, теперь интерпретатор запрещает некоторые комбинации операторов при наличии во вложенных блоках кода свободных переменных. В листинге 8.59 приведены три таких случая:

Листинг 8.59. Запретные комбинации

```
from __future__ import nested_scopes

def example1(v):
    def bad():
        print v
    bad()
    del v  # нельзя удалять имена, используемые во вложенном блоке

def example2(x):
    from string import *  # нельзя делать import *
    def b(y):
       print join, b      # если во вложенном блоке
                          # есть свободные переменные

def example3(x):
    def b(y):
       print join, b
    exec x                # нельзя делать exec без явного
                          # указания пространств имен
```

Все эти запреты и ранее были описаны в документации Python, но теперь они стали обязательными. Следует сказать, что приведенные выше примеры функций `example1()`, `example2()`, `example3()` являются примерами запутанного кода, в котором трудно найти ошибку. Предлагаемые ниже исправления уже не подпадают под запреты и делают код более наглядным и контролируемым, см. листинг 8.60.

Листинг 8.60. Исправление кода

```
def example1(v):
    def bad(v1):
```

```
        print v1
    bad(v)
    del v

def example2(x):
    from string import join
    def b(y):
       print join, b

def example3(x):
    def b(y):
       print join, b
    exec x in locals()
```

Основные *проблемы совместимости кода*, возникающие со старым кодом при интерпретации новым Python, связаны именно с указанными выше случаями. К счастью, при правильном стиле программирования эти проблемы возникают крайне редко (или не возникают вообще). Как сказано в PEP 227, на 200 000 строк кода в трех рассмотренных проектах на Python было найдено всего четыре проблемы совместимости.

Еще раз подчеркнем, что пространств имен по-прежнему три (глобальное, локальное и встроенное). Просто теперь из каждой точки кода они видны по-другому, более широко.

8.8. Модули и пакеты

Только очень небольшие программы можно успешно писать и, главное, сопровождать, не разбивая на *модули*. В отдельный модуль обычно выделяется некоторая относительно независимая часть программы, часто посвященная отдельному объекту. Все доступные в Python библиотеки реализованы в виде модулей (часть из этих модулей, например, `sys`, являются встроенными). Программа может подгружать нужные модули или даже отдельные объекты из модулей с помощью операторов `import` и `from-import`. Эти операторы создают объект-модуль на основе файлов с описаниями модулей, найденных на пути поиска. Путь поиска модулей в системе может быть свой, но обычно включает в себя текущий каталог. За исключением трансляции модуля, все действия с модулем как объектом производятся только в оперативной памяти. Трансляция же происходит только если файл модуля (name.py) старше транслированного файла (name.pyc или name.pyo) или транслированный файл отсутствует.

Модуль можно запустить как обычную программу (или можно считать, что обычная программа — специальный модуль по имени `__main__`). При этом можно даже выполнять некоторый код только для случая, когда модуль запускается как отдельная программа.

Листинг 8.61. Файл mymodule.py. Модуль `mymodule.py`

```
"""Модуль предоставляет функции по суммированию чисел в строке"""
from string import split
import re
# предварительно скомпилируем регулярное выражение
fmt_re = re.compile("^\d+$")
def sum_string(s):
  S = 0
  for n in split(s):      # для всех слов строки:
    if fmt_re.match(n):   # если слово -- набор цифр:
      S = S + int(n)      # добавить число к сумме
  return S
print "Исполнение кода модуля."
# если файл модуля интерпретируется как основная программа:
if __name__ == "__main__":
  print "Тест модуля mymodule"
  print __doc__
  if sum_string("1 : 2 - 3     4   \n") == 10:
     print "Тест пройден"
  else:
     print "Тест провален"
```

Листинг 8.62. Файл mainprog.py. Главная программа

```
import mymodule
print "__doc__:", mymodule.__doc__
print "__file__:", mymodule.__file__
print "__name__:", mymodule.__name__
print "sum(1 2 3 4):", mymodule.sum_string("1 2 3 4")
```

Результат выполнения `mymodule.py`:

```
Исполнение кода модуля.
Тест модуля mymodule
Модуль предоставляет функции по суммированию чисел в строке
Тест пройден
```

Результат выполнения `mainprog.py`:

```
Исполнение кода модуля.
__doc__: Модуль предоставляет функции по суммированию чисел в строке
__file__: mymodule.pyc
__name__: mymodule
sum(1 2 3 4): 10
```

Модули могут содержать другие модули, но при этом не должны импортировать друг друга. Модули могут быть организованы в *пакеты (packages)*.

Пакет хранится в отдельном каталоге на пути поиска модулей. Этот каталог содержит файл __init__.py, который нужен для инициализации модулей пакета. Для примера можно рассмотреть пакет xml (файлы с расширениями pyc и pyo не показаны):

Листинг 8.63. Дерево каталогов пакета `xml`

```
-- xml
   |-- __init__.py
   |-- dom
   |   |-- __init__.py
   |   |-- minidom.py
   |   `-- pulldom.py
   |-- parsers
   |   |-- __init__.py
   |   `-- expat.py
   `-- sax
       |-- __init__.py
       |-- _exceptions.py
       |-- expatreader.py
       |-- handler.py
       |-- saxutils.py
       `-- xmlreader.py
```

Таким образом, пакет `xml` состоит из трех *подпакетов (sub-packages)*: `dom`, `parsers` и `sax`. У каждого подпакета свой `__init__.py`.

Если заглянуть в `__init__.py` пакета `xml`, среди прочего можно разглядеть следующее присваивание:

```
__all__ = ["dom", "parsers", "sax"]
```

Этот список говорит интерпретатору о том, какие модули (или подпакеты) следует импортировать по команде `from xml import *`.

Добраться до модулей пакета можно следующими способами (на примере того же `xml`):

- `import xml.dom.minidom` — загружает модуль `xml.dom.minidom` и делает доступным класс `Document` через `xml.dom.minidom.Document`;
- `from xml.dom import minidom` — то же, но теперь `minidom` доступен как `minidom`, а `Document` — как `minidom.Document`;
- `from xml.dom.minidom import Document` — делает доступным класс `Document` как `Document`.

При загрузке модуля из некоторого пакета, этот пакет тоже загружается и инициализируется. Например, при загрузке `xml.dom.minidom` загружаются `xml` (исполняется код из `xml/__init__.py`) и `xml.dom` (исполняется `xml/dom/__init__.py`).

В указанной выше "точечной" нотации, первые элементы должны быть пакетами, а последний может быть как пакетом, так и модулем. В нашем примере `xml` и `dom` — пакеты, а `minidom` — вложенный в них модуль.

Если посмотреть внутрь одного из модулей пакета, мы увидим, что внутри модуля ссылаться на лежащие в том же каталоге модули и пакеты можно просто по имени. Так в `xml/sax/__init__.py` выглядит импорт из модуля `xml.sax.handler`:

```
from handler import ContentHandler, ErrorHandler
```

Пакеты в отдельные объекты не выделяются — это те же объекты-модули. По сравнению с обычными модулями, объект-модуль для пакета имеет атрибут `__path__`, в котором хранится информация о пути каталогу пакета. В некоторых пакетах манипуляция с `__path__` при инициализации необходима для того, чтобы задействовать платформо-зависимые модули.

В случае, когда на пути поиска в одном каталоге содержится и пакет, и модуль с тем же именем, будет загружен пакет. Если все-таки необходимо, чтобы первым находился модуль, нужно поместить его в каталог, находящийся ранее каталога, в котором хранится пакет, в `sys.path`.

Пакеты и модули можно хранить не только в каталогах, указанных в `sys.path`. В этих каталогах можно оставить символическую ссылку (ярлык) на каталог, в котором находится пакет. Другой способ заключается в хранении в одном из каталогов `sys.path` (желательно, `site-packages`) файла с расширением pth, в котором указаны пути к каталогам с модулями. Второй путь работает с теми модулями, которые не структурированы в пакеты, но хранятся в отдельных каталогах.

8.9. Оптимизация программ

На момент написания этой книги основным недостатком языка Python, с точки зрения автора книги, является его невысокая производительность. Конечно, от интерпретируемого языка сложно требовать производительности компилируемых языков, и требовательные к процессорному времени задачи лучше писать, скажем, на C. Тем не менее, и программы на Python можно (подчас значительно!) *оптимизировать*, использовав приведенные в этом разделе рекомендации. Эти рекомендации основаны на документе "The whole Python FAQ", который находится в Internet по адресу **http://www.python.org/doc/FAQ.html** и постоянно обновляется, вбирая опыт сотен пользователей.

К Python применимы общие для языков программирования методики оптимизации. Например, ввод и вывод с бóльшим размером буфера и более крупными порциями. В "The whole Python FAQ" утверждается, что CGI-сценарий, написанный на Python, заработает заметно быстрее, если осуществить вывод сформированной Web-страницы единой порцией. Однако для тех, кто перешел на Python с Pascal или C, следует знать, что их подход к написанию программ в том же стиле — не всегда самое эффективное решение с точки зрения Python, и им будет полезно прислушаться к данным ниже рекомендациям.

Прежде всего, необходимо упростить программу. Меньше кода — меньше команд транслировать и исполнять интерпретатору, меньше темных закоулков. И, как результат, — меньше ошибок.

Узнать, где именно тратится процессорное время, можно с помощью *профайлера (см. разд. 9.11.2)*. Этот инструмент покажет вам те места программы, которые стоит оптимизировать.

Вот некоторые из рекомендаций.

- Там, где это уместно, лучше заменять циклы на операции обработки последовательностей, например:

```
s2 = []
for i in s1:
  s2.append(string.split(s1))
```

  можно с успехом заменить на:

```
s2 = map(string.split, s1)
```

- Повторные конкатенации ряда больших строк стоит заменить на `string.join` списка строк. Вместо:

```
s2 = ""
for i in s1:
    s2 = s2 + f(i) + "\n"
```

можно написать:

```
import string
s2 = string.join(map(f, s1)+[""], "\n")
```

- Используйте вместо конкатенации больших строк операцию форматирования, т. е. вместо:

```
print s1 + ": " + s2 + bigs3 + "(" + s4 + ")" + bigs5
```

лучше воспользоваться:

```
print "%s: %s%s(%s)%s" % (s1, s2, bigs3, s4, bigs5)
```

или даже:

```
print "%(s1)s: %(s2)s%(bigs3)s(%(s4)s)%(bigs5)s" % vars()
```

- Пользуйтесь встроенными методами сортировки списков: `list.sort()`.
- Считается, что вызов функции (метода) и обращение к глобальной переменной в Python — "дорогая" операция. В связи с этим лучше избегать множества вызовов функций из больших или глубоко вложенных циклов.
- В критичных циклах рекомендуется убирать все выражения, которые в цикле не меняются, но тем не менее вычисляются вновь и вновь:

```
for key in keys:
  dict[key] = dict.get(key, 0) + 1
```

рекомендуется заменить на:

```
dict_get = dict.get
for key in keys:
  dict[key] = dict_get(key, 0) + 1
```

- В некоторых случаях можно вынести вычисление констант, используемых в функциях, в аргументы со значением по умолчанию. В этом случае эти константы будут вычислены всего один раз:

```
import math
def Circle(D):
  return 2 * math.pi * D
```

можно записать как:

```
import math
def Circle(D,
           pi2 = 2*math.pi):
  return pi2*D
```

Конечно, этот прием, оптимизирующий исполнение программы, затрудняет ее чтение и может запутать пользователей функции, поэтому его стоит применять лишь для внутренних функций.

8.10. О стиле программирования

Стиль программирования — дополнительные ограничения, накладываемые на структуру и вид программы группой совместно работающих программистов с целью получения удобных для применения, легко читаемых и эффективных программ. Основные ограничения на вид программы дает синтаксис языка программирования, и его нарушения, очевидно, приводят к синтаксическим ошибкам. Нарушения в стиле не приводят к ошибкам, но тем не менее как отдельные программисты, так и целые коллективы сознательно берут на себя труд поддерживать тот или иной стиль для упрощения разработки, отладки и сопровождения.

Стиль программирования затрагивает все аспекты написания кода на языке программирования:

- задание имен объектов;
- оформление функций, методов, классов, модулей и их документирование;
- декомпозиция (разбиение) программы на модули определенного размера;
- включение отладочной информации;
- применение тех или иных функций (методов) исходя из предполагаемого уровня совместимости разрабатываемой программы с различными компьютерными платформами;
- ограничение используемых функций из соображений безопасности.

Для языка Python его автор разработал стиль, которым пользуется уже несколько лет и рекомендует всем. С оригинальным текстом "Python Style Guide" можно ознакомиться на **http://www.python.org/doc/essays/styleguide.html**. Ниже мы рассмотрим наиболее существенные положения этого стиля.

- Рекомендуется использовать отступы в 4 символа.
- Длина физической строки не должна превышать 79 символов.
- Длинные логические строки лучше разбивать неявными методами (внутри скобок), но и явные методы (с помощью "\") вполне уместны. Отступы строк продолжения лучше выравнивать по скобкам или по первому операнду в предыдущей строке:

```
def draw(figure, color="White", border_color="Black",
         size=5):
    if color == border_color or \
       size == 0:
        raise "Bad figure"
    else:
       _draw(size, size, (color,
                          border_color))
```

Текстовый редактор Emacs в режиме `python-mode` автоматически делает необходимые отступы.

- Пустые строки тоже можно использовать для улучшения читаемости программы. Одиночные пустые строки можно использовать внутри определений функций для выделения в них логических блоков. Рекомендуется отделять описания функций от описаний классов двумя пустыми строками, определения функций и методов отделять пустой строкой. Если пустая строка применяется для отделения методов класса, после строки с оператором `class` также должна идти пустая строка. Дополнительной пустой строкой можно выделить группы функций, теснее связанных между собой. Но в группе нескольких функций с однострочным определением можно не использовать пустые строки.

- Автор Python не рекомендует ставить пробелы сразу после открывающей скобки или перед закрывающей, перед запятой, точкой с запятой, перед открывающей скобкой при записи вызова функции или индексного выражения. Также не рекомендуется ставить более одного пробела вокруг "=" в присваиваниях. Пробелы вокруг "=" не ставятся в случае, когда он применяется для указания значения по умолчанию в определении функции или при задании именованных аргументов.

  Другие рекомендации по использованию пробелов включают использование одиночных пробелов вокруг низкоприоритетных операций сравнения и оператора присваивания. Пробелы вокруг более приоритетных операций ставятся на ваш вкус, но в одинаковом числе слева и справа от знака операции.

- Несколько рекомендаций касаются написания комментариев. Комментарии должны точно отражать актуальное состояние кода. (Поддержание актуальных комментариев должно быть приоритетной задачей!). После коротких комментариев можно не ставить точку, тогда как длинные лучше писать по правилам написания текста. Автор Python отдельно обращается к неанглоязычным программистам с просьбой писать комментарии на английском, если есть даже небольшая вероятность того, что код будут читать специалисты, говорящие на других языках.

  Комментарии к следующему далее фрагменту кода следует писать с применением тех же отступов, что и комментируемый код. После "#" идет одиночный пробел. Абзацы можно отделять строкой с "#" на том же уровне. Блочный комментарий можно отделить пустыми строками от окружающего кода.

  Комментарии, относящиеся к конкретной строке, следует использовать не так часто. Символ "#" должен отстоять от комментируемого оператора как минимум на два пробела. Хороший комментарий не перефразирует программу, а содержит дополнительную информацию о действии программы в терминах предметной области.

- В идеале все модули, классы, функции и методы, предназначенные для экспорта из модуля, должны иметь строки документации, как минимум описывающие способ их применения, входные и выходные параметры. Строка документации для отдельной программы должна объяснять используемые для ее запуска ключи, смысл аргументов и переменных среды, а также другую подобную информацию.

  Для строк документации рекомендуется везде использовать утроенные кавычки (`"""`).

  Однострочная документация пишется в императиве, как команда: "делай это", "возвращай то-то и то-то".

  Многострочная документация содержит расширенное описание модуля, функции, класса. Она будет смотреться лучше, если текст будет написан с тем же отступом, что и начало строки документации.

  Документация для модуля должна перечислять экспортируемые функции, классы, исключения и другие объекты, по одной строке на объект.

  Строка документации для функции или метода должна кратко описывать действия функции, ее входные параметры и возвращаемое значение, побочные эффекты и возможные исключения (если таковые есть). Должны быть обозначены необязательные аргументы и аргументы, не являющиеся частью интерфейса.

  Документация для класса должна перечислять публичные (public) методы и атрибуты объектов, давать рекомендации по использованию класса в качестве базового для других классов. Если класс является подклассом, необходимо указать, какие методы полностью заменяют, перегружают (override), а какие используют, но расширяют (extend) соответствующие методы суперкласса. Необходимо указать и другие изменения по сравнению с суперклассом.

- Контроль версий поднимает качество процесса создания программного обеспечения. Для этих целей часто используются RCS или VCS. "Python Style Guide" рекомендует записывать `$Revision: 4.0 $` в переменную с именем `__version__`, а другие данные заключать в комментарии `"#"`.

Сейчас есть несколько более или менее широко распространенных правил именования объектов, и программисты, конечно, вольны выбирать тот, который принят в рамках их организации или проекта. Автор Python рекомендует придерживаться данных ниже правил для именования различных объектов, т. к. в этом случае вас поймет любой программист, использующий Python.

- Имена модулей лучше давать маленькими буквами, как например, `shelve`, `string`, либо делать первые буквы слов прописными, как в `StringIO`, `UserDict`. Кроме того, имена модулей, написанных на C,

обычно начинаются с подчеркивания "_", а соответствующие им высокоуровневые "обертки" — с прописных букв, как `_tkinter` и `Tkinter`.

- Ключевые слова нельзя использовать в качестве имен, однако, если вам все-таки необходимо воспользоваться этим именем, добавьте одиночное подчеркивание на конце, например: `class_`.
- Классы обычно называют, выделяя первые буквы слов прописными, как в `Tag` или `HTTPServer`.
- Имена исключений обычно содержат в своем составе слово "`error`". Встроенные модули пишут это слово с маленькой буквы (как `os.error`), но могут писать и с прописной: `distutils.DistutilsModuleError`.
- Функции, экспортируемые модулем, могут именоваться по-разному. Можно давать с прописных букв имена наиболее важных функций, а вспомогательные писать строчными.
- Имена глобальных переменных (если вы таковые используете) лучше снабжать подчеркиванием, чтобы они не экспортировались из модуля оператором `from-import`.
- Для имен методов можно пользоваться теми же правилами, что и для функций.
- Имена констант (то есть, имен, которые не должны переопределяться) лучше записывать прописными буквами, например: `RED`, `GREEN`, `BLUE`.
- При работе с Python необходимо учитывать, что интерпретатор считает некоторые классы имен специальными *(см. статью "Идентификаторы" в разд. 8.3.1)*.

Конечно, выдержки из документа, описывающего "официальный" стиль для языка Python, — всего лишь рекомендованная основа для выработки своего стиля или адаптации "фирменного" стиля к языку Python. Главное в этом процессе, чтобы за формой не было потеряно красивое и эффективное содержание — качественное алгоритмическое наполнение ваших программ.

В каждом языке программирования за время его существования и развития складываются определенные представления о том, как следует реализовывать те или иные фрагменты кода. Такие устоявшиеся конструкции по аналогии с естественными языками принято называть *идиомами*. В языке Python тоже сложилась своя идиоматика, которая накапливается при общении с другими Python-программистами или просмотре исходных кодов классических образцов (например, стандартной библиотеки). В идиомах обычно отражены наиболее эффективные пути решения типичных микропроблем. Например, построчное чтение файла `f` можно реализовать несколькими способами, но до появления в файловом объекте Python 2.1 метода `xreadlines()` обычно использовалась следующая идиома:

```
f = open("file", "rt")
while 1:
```

```
  line = f.readline()
  if not line:  # если конец файла
    break
  # некоторые действия с line
f.close()
```

Владение идиомами позволяет не только легче читать чужой код, но и облегчает написание своего, ведь код часто состоит из подобных микроситуаций, в которых применима та или иная идиома.

8.11. Направления развития языка

Язык Python не является законченным программным продуктом. Его разработчики постоянно вносят в язык изменения и дополнения, стараясь при этом не нарушать основных "заповедей", выраженных Тимом Питерсом (Tim Peters) и называемых иногда "Zen of Python" по аналогии с заповедями дзен-буддизма, вот они:

"Красивое лучше уродливого. Явное лучше неявного. Простое лучше сложного. Сложное лучше усложненного. Плоское лучше вложенного. Разреженное лучше плотного. Читаемость важна. Частные случаи недостаточно частные, чтобы нарушать правила. Хотя практичность превыше чистоты, ошибки не должны подавляться, если только явно не указано обратное. В случае неопределенности не поддавайся соблазну угадывания. Должен быть один, а лучше только один, очевидный путь решения, хотя этот путь может быть неочевиден с первого раза, если ты не голландец (не только не ван Россум, здесь имеется в виду и другое значение слова «голландец» в английском языке, — «странный человек, оригинал»). Сейчас лучше, чем никогда, хотя лучше никогда, чем прямо сейчас. Если реализацию идеи сложно объяснить, это плохая идея. Если реализацию идеи легко объяснить, возможно, идея хорошая. Пространство имен — одна из звучных идей, давайте сделаем больше таких".

За время написания этой книги вышел Python 2.1, а о Python 2.2 стало почти достоверно известно, что в него будет включено, т. к. на прилагаемом компакт-диске уже содержится α-версия. Ниже сделана попытка описать самые главные нововведения и попытаться предугадать, чего же ждать от разработчиков Python в будущем.

Начиная с Python 2.2, в языке Python в явном виде появляются *итераторы (iterators)*. Если раньше для эмуляции (потенциально) бесконечной последовательности приходилось хитрить с методом `__getitem__()` *(см. статью*

"Оператор for" в разд. 8.5.2), то теперь итератор можно задать в явном виде, как в следующем примере:

Листинг 8.64. Файл fib2.py. Итератор чисел Фибоначчи (Python 2.2)

```
class Fib:
  """Последовательность Фибоначчи"""
  def __init__(self, max):
    self.max = max
    self.n, self.a, self.b = 0, 0, 1
  def __iter__(self):
    return self
  def next(self):
    if self.n < self.max:
      a = self.a
      self.n, self.a, self.b = (self.n + 1, self.b, self.a + self.b)
      return a
    else:
      raise StopIteration

for i in Fib(10):
  print i,
```

Результат выполнения:

```
0 1 1 2 3 5 8 13 21 34
```

Подобное описание класса ни у кого не вызывает сомнений в назначении его экземпляров.

Кроме итераторов, в Python 2.2 появляются объекты, которые документация называет *простыми генераторами (simple generators)*. Генераторы образуются при вызове специальных функций, определяемых с помощью оператора `def`. Функции для порождения генераторов отличает от обычных функций наличие оператора `yield` (давать), а также обязательное отсутствие аргумента у оператора `return`. Функции, порождающие генератор, выполняются только до встречи оператора `yield`, после чего контекст исполнения кода запоминается в возвращаемом объекте-генераторе. Этот объект ведет себя как итератор, который при каждом запросе продолжает исполнение тела породившей его функции до очередного оператора `yield`. Этот оператор и дает очередное значение. Если вместо `yield` встретился `return` или поток управления дошел до конца тела функции, генератор возбуждает исключение

StopIteration. Исключение является сигналом того, что больше значений не будет, цикл for завершается.

В листинге 8.65 мы делаем то же самое, что и в предыдущем, но уже с использованием генератора:

Листинг 8.65. Файл fib3.py. Генератор чисел Фибоначчи (Python 2.2)

```
from __future__ import generators
def Fib(max):
  a, b = 0, 1
  for i in xrange(max):
    yield a
    a, b = b, a + b
return
for i in Fib(10):
  print i,
```

Программа стала короче, а результат остается тем же:

```
0 1 1 2 3 5 8 13 21 34
```

Таким образом, в Python 2.2 появился механизм, позволяющий менее расточительно использовать оперативную память. По сути, введение функций xrange() и метода xreadlines() преследовало ту же цель. Теперь же появляется возможность пользоваться генераторами без дополнительных классов. Скорее всего, после введения генераторов обрабатывать длинные последовательности данных (не загружая их в память целиком) станет еще проще.

Кроме того, в будущем нас ждет перемена в семантике операции "/". До сих пор Python использовал принятую в языке C семантику для этой операции, когда результат не выходит за пределы наиболее информативного типа. Теперь же старая операция будет доступна как "//", а у программистов будет время (до версии Python 3.0) переписать свои программы с использованием новой операции "/". Следующий пример показывает суть грядущего изменения:

```
>>> print 3/4
0
>>> from __future__ import division
>>> print 3/4
0.75
>>> print 3//4
0
```

Скоро в языке Python будут стерты различия между типами и классами, т. к. можно будет наследовать от встроенных типов, что и демонстрируется в следующем примере:

```
>>> import types
>>> class Int(types.IntType):
...   def __str__(self):
...     return "%08i" % self
...
>>> k = Int(5)
>>> l = k + 3
>>> print k, l
00000005 8
```

Мы рассмотрели лишь самые масштабные изменения: мелких изменений гораздо больше. В результате Python становится еще более понятным и логичным.

Глава 9



Основные стандартные модули

Глава служит путеводителем по наиболее часто используемым модулям. Рассматриваемые здесь модули входят в поставку Python на различных платформах. Для упрощения поиска нужной функции (класса, модуля) сделана попытка сгруппировать модули с близкими функциями в отдельные разделы.

9.1. Встроенная помощь: модуль *pydoc*

В Python 2.1 появился долгожданный модуль для интерактивного доступа к документации. Следующие примеры демонстрируют некоторые возможности модуля `pydoc`, которые могут помочь начинающим.

Листинг 9.1. Получение помощи по модулю `quopri`

```
>>> from pydoc import help
>>> help('quopri')
Help on module quopri:

NAME
    quopri - Conversions to/from quoted-printable transport encoding as
per RFC-1521.

FILE
    /usr/lib/python2.1/quopri.py

DATA
    ESCAPE = '='
    HEX = '0123456789ABCDEF'
    MAXLINESIZE = 76
    __all__ = ['encode', 'decode']
    __file__ = '/usr/lib/python2.1/quopri.pyc'
```

```
    __name__ = 'quopri'
(END)q
>>>
```

Листинг 9.2. Получение помощи по функции `string.maketrans`

```
>>> from pydoc import help
>>> help('string.maketrans')
Help on built-in function maketrans in string:

maketrans(...)
    maketrans(frm, to) -> string

    Return a translation table (a string of 256 bytes long)
    suitable for use in string.translate.  The strings frm and to
    must be of the same length.
(END)q
```

Помощь может работать и в специальном режиме подсказки:

Листинг 9.3. Режим подсказки

```
>>> help

Welcome to Python 2.1!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://www.python.org/doc/tut/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

То есть, на приглашение help> вы можете вводить все то же самое, что и в аргументе функции help(). Получить список установленных модулей можно, введя modules, список ключевых слов — по keywords. Также можно ввести topics, чтобы получить темы, по которым возможна помощь.

Листинг 9.4. Получение списка модулей с "http" в имени

```
help> modules http

Here is a list of matching modules.  Enter any module name to get more
help.

BaseHTTPServer - HTTP server base class.
CGIHTTPServer - CGI-savvy HTTP Server.
SimpleHTTPServer - Simple HTTP Server.
httplib - HTTP/1.1 client library
test.test_httplib

help>
```

Завершить работу с подсказкой можно, набрав **quit** и нажав клавишу <Enter>.

Для правильной работы topics и keywords необходимо, чтобы в системе была установлена документация по Python. Указать место расположения файлов помощи (в формате HTML) можно установкой переменной окружения PYTHONDOCS до импортирования модуля pydoc:

```
>>> import os
>>> # Это только пример! В вашей системе документация может
>>> # находиться в другом месте.
>>> os.environ['PYTHONDOCS'] = '/usr/doc/python-2.1/'
>>> from pydoc import help
```

Заметьте, что помощь может быть оказана не только по стандартным модулям. Если ваши собственные модули, классы, методы и функции имеют строки документации, вы можете получить помощь и по ним, как это продемонстрировано в листинге 9.5.

Листинг 9.5. Помощь по документированному классу и методу класса

```
>>> class Object:
...     """My object"""
...     def myfun(self):
```

```
...     """My function"""
...
>>> help(Object)
Help on class Object in module __main__:

class Object
 |  My object
 |
 |  myfun(self)
 |      My function

>>> help(Object().myfun)
Help on method myfun:

myfun(self) method of __main__.Object instance
    My function
```

Помимо применения `pydoc` в качестве модуля, вы можете использовать программу `pydoc`, основанную на этом модуле, из командной строки. Ниже приведен сокращенный перевод экрана помощи этой программы:

Листинг 9.6. Экран помощи программы `pydoc`

```
pydoc - помощь по Python

pydoc имя ...
  Показать документацию по функции, модулю, пакету, классу, и т. п.
с заданным именем.

pydoc -k ключевое слово
  Найти модули, в описании которых встречается данное ключевое слово.

pydoc -p порт
  Стартует HTTP-сервер на заданном порту на локальной машине.

pydoc -g
  Вызывает графический интерфейс для поиска и показа документации.
Для показа документации используется Web-браузер.
```

`pydoc -w имя ...`

Записывает документацию по заданному имени в формате HTML в файл, находящийся в текущем каталоге.

9.2. Сервисы периода исполнения

Эти сервисы осуществляются самыми разными модулями, предоставляющими функции, относящиеся к взаимодействию с интерпретатором Python и, до некоторой степени, средой, в которой выполняется интерпретатор.

9.2.1. Модуль *sys*

Модуль `sys` дает доступ к разнообразным переменным, которые используются или поддерживаются непосредственно интерпретатором Python или тесно с ним связаны. Эти переменные и функции приведены в табл. 9.1.

Таблица 9.1. *Переменные и функции модуля* `sys`

Объект	Описание
`sys.argv`	Список аргументов командной строки. Элемент `sys.argv[0]` содержит название сценария, `"-c"`, если сценарий был запущен из командной строки по ключу `-c`. Если имя сценария не передано интерпретатору, `argv[0]` имеет нулевую длину
`sys.byteorder`	Порядок байтов в слове, принятый на данной платформе. Может быть `'big'` (старшие-младшие) или `'little'` (младшие-старшие)
`sys.builtin_module_names`	Кортеж встроенных (на этапе компиляции) модулей. Обычно содержит `__builtin__`, `sys` и некоторые другие
`sys.copyright`	Строка с записью принадлежности авторских прав
`displayhook(obj)`	Печатает объект на стандартный вывод, а также помещает его в `__builtin__._`. Используется в интерактивном режиме для вывода результатов. Может быть переопределена. Оригинал находится в `sys.__displayhook__`. Функция появилась в Python 2.1
`sys.dllhandle`	Целое число с манипулятором Python DLL (только для Windows)
`sys.exc_info()`	Возвращает кортеж с информацией об исключении `(exc_type, exc_value, exc_traceback)`, где `exc_type` — объект, идентифицирующий исключение, `exc_value` хранит дополнительную информацию, а `exc_traceback` содержит трассировочный объект с информацией о месте, в котором произошло исключение

Таблица 9.1 *(продолжение)*

Объект	Описание
`sys.exec_prefix`	Префикс каталога с платформо-зависимыми файлами, относящимися к интерпретатору Python
`sys.excepthook(exc_type, exc_value, exc_traceback)`	Обрабатывает дошедшее до интерпретатора исключение. Значения аргументов объяснены выше. Действие заключается в выводе трассировочной информации на стандартный вывод ошибок. Функцию можно переопределить для своих нужд. Оригинал в `sys.__excepthook__`. Появилась в Python 2.1
`sys.executable`	Путь к интерпретатору Python (на платформах, где это имеет смысл)
`sys.exit([arg])`	Выход из интерпретатора. Параметр `arg` задает целое число — код выхода (при успешном завершении программы он равен нулю), но также может быть любым другим объектом (в этом случае этот объект будет напечатан в файл sys.stderr)
`sys.exitfunc`	Переменная, которой можно присвоить функцию для исполнения при завершении программы (функция не должна иметь параметров). Эта функция не будет вызвана, если программа завершена по сигналу, из-за внутренней ошибки в интерпретаторе или при вызове функции `os._exit()` *(см. также разд. 9.2.2)*
`sys.getrefcount(obj)`	Функция возвращает количество ссылок на объект `obj`
`sys.getrecursionlimit()`	Функция возвращает целое значение — глубину максимально возможной рекурсии *(см.* `setrecursionlimit()` ниже)
`sys.hexversion`	Версия интерпретатора Python. Гарантированно растет от версии к версии
`sys.last_type`, `sys.last_value`, `sys.last_traceback`	Эти три переменные получают свое значение в интерактивном режиме в случае, когда возникшее исключение не обрабатывается пользовательской программой. Например, можно использовать `import pdb; pdb.pm()` для post mortem отладки
`sys.maxint`	Максимальное целое значение
`sys.modules`	Словарь, устанавливающий соответствие между именами загруженных модулей и модулями
`sys.path`	Список строк, указывающих пути к каталогам с модулями
`sys.platform`	Используемая компьютерная платформа, например, `"linux-i386"` или `"sunos5"`

Таблица 9.1 *(окончание)*

Объект	Описание
`sys.prefix`	Префикс каталога с платформо-независимыми файлами установки Python
`sys.ps1, sys.ps2`	Переменные, в которых можно задать отличные от `">>>"` и `"..."` приглашения интерпретатора. В качестве значений можно передавать не только строки, но и другие объекты. В этом случае к ним будет применяться функция `str()`. См. примеры ниже, где в приглашении интерпретатора показывается время
`sys.setcheckinterval (int)`	Функция, задающая интервал проверки различных асинхронных событий (переключение потоков исполнения или обработки сигналов). По умолчанию события проверяются и обрабатываются каждые 10 элементарных инструкций
`sys.setprofile(func)`	Задание функции для профайлера. Функция вызывается при каждом входе или выходе из функции. Аналогична `sys.settrace()`
`sys.setrecursionlimit (int)`	Устанавливает новое значение для глубины максимально возможной рекурсии
`sys.settrace(func)`	Задание функции для отладки. Функция будет вызываться для каждой строки кода
`sys.stdin, sys.stdout, sys.stderr`	Файловые объекты для стандартного ввода, вывода и вывода ошибок
`sys.__stdin__, sys.__stdout__, sys.__stderr__`	Оригинальные файловые объекты для стандартного ввода, вывода и вывода ошибок
`sys.version`	Строка с указанием версии интерпретатора, используемой платформы, даты компиляции и названия компилятора
`sys.version_info`	Информация о версии в виде кортежа из пяти значений. Первые три значения соответствуют главному (major), второстепенному (minor) и микрономеру версии Python, затем следует статус выпуска (`'alpha'`, `'beta'`, `'candidate'` или `'final'`). И, наконец, последнее значение соответствует серии

Пример в листинге 9.7 показывает одно из наиболее типичных применений модуля `sys`:

Листинг 9.7. Программа для печати элементов командной строки

```
import sys
print "Аргументы командной строки:"
```

```
for arg in sys.argv:  # все аргументы командной строки
  sys.stdout.write(arg+"\n")  # выводить на стандартный вывод
sys.exit("bye-bye")
```

Еще один пример связан с интерактивной оболочкой Python. В этом примере в качестве приглашения интерпретатора используется объект, строковое представление которого вычисляется каждый раз с использованием текущего времени:

```
>>> import sys, time
>>> class show_time:
...   def __str__(self):
...     return time.strftime("%H:%M>>> ", time.localtime(time.time()))
...
>>> sys.ps1 = show_time()
21:20>>> print sys.ps2
...
21:21>>>
```

9.2.2. Модуль *atexit*

Модуль `atexit` предоставляет возможность вызывать произвольные функции при выходе из программы. В модуле определена функция `register()`, которая служит для регистрации пользовательской функции, вызываемой при завершении программы (и ее аргументов). Функции вызываются в порядке, обратном порядку регистрации.

```
atexit.register(func[, *targs][, **kargs])
```

где `func` — функция, которую нужно запустить; `targs` — (необязательные) позиционные аргументы; `kargs` — (необязательные) именованные аргументы.

```
from atexit import register
def x1():
    print "x1: пока!"
def x2(name):
    print "x2: пока, %s!" % name
def x3(name="неизвестный", place="откуда-то"):
    print "x3: пока, %(name)s из %(place)s!" % vars()
register(x1)
register(x2, "программист")
register(x3, name="великий программист", place="далекой страны")
```

Результат выполнения:

```
x3: пока, великий программист из далекой страны!
x2: пока, программист!
x1: пока!
```

9.2.3. Модуль *copy*

Модуль `copy` предоставляет функции для *поверхностного* и *глубокого копирования* объектов. Разница между этими двумя видами копирования существенна только для составных объектов. При поверхностном копировании создается `obj1`, подобный `obj`, и наполняется ссылками на объекты из `obj` (по мере возможности). Глубокое копирование рекурсивно копирует все составные части объекта. Следующие функции обеспечивают *поверхностное* и *глубокое копирование*:

- `copy.copy(obj) -> obj1`
- `copy.deepcopy(obj) -> obj1`

В случае возникновения ошибок при копировании возбуждается исключение `copy.error`.

```
import copy
x1 = copy.copy(x)          # поверхностная копия
x1 = copy.deepcopy(x)      # глубокая копия
```

В некоторых случаях программисты, привыкшие к тому, что присваивание приводит к поверхностному копированию объектов, могут совершить совсем неочевидные ошибки, программируя на Python. В приведенном ниже примере вместо ожидаемой инициализации модели двумерного массива получается несколько иная структура. Желаемую же структуру можно получить только с помощью функции копирования. Кстати, если потребуется трехмерный массив, придется использовать уже глубокую копию.

Листинг 9.8. Неправильный подход к построению двумерного массива

```
>>> i = 1
>>> a=[[i]*3]*4
>>> a
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]
>>> a[1][1] = 2
>>> a
[[1, 2, 1], [1, 2, 1], [1, 2, 1], [1, 2, 1]]
```

Размноженные списки — один и тот же объект, поэтому изменение любого из них неожиданно сказывается на других. Решить проблему можно с помощью поверхностного копирования:

Листинг 9.9. Исправленный вариант построения двумерного массива

```
Пример:
>>> import copy
>>> i = 1
>>> a = [copy.copy([i]*3) for k in range(4)]
>>> a
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]
>>> a[1][1] = 2
>>> a
[[1, 1, 1], [1, 2, 1], [1, 1, 1], [1, 1, 1]]
```

В этом примере для более компактной записи приведено списковое включение *(см. статью "Задание кортежей, списков, словарей" в разд. 8.3.1)*.

Применяя `copy.deepcopy()`, следует помнить о некоторых опасностях. Во-первых, рекурсивное копирование может затронуть данные, которые должны быть общими даже для копий. Эта проблема решается использованием в пользовательских классах специальных методов, управляющих копированием. Во-вторых, объекты могут содержать циклические ссылки. Python избегает проблем с зацикливанием, храня таблицу уже скопированных объектов.

Приведенные в табл. 9.2 методы, заданные в определяемом пользователем классе, позволяют управлять как глубоким копированием, так и операцией модуля `pickle` *(см. разд. 9.2.7)*.

Таблица 9.2. *Методы для управления копированием*

Метод	Описание
`__getinitargs__()`	Должен возвращать кортеж из аргументов, который будет подаваться на вход метода `__init__()` при создании объекта
`__getstate__()`	Должен возвращать словарь — состояние объекта (если класс не имеет этого метода, то используется `__dict__` копируемого объекта)
`__setstate__(state)`	Должен присваивать ранее сохраненное состояние объекту

Листинг 9.10. Пример класса, управляющего своим копированием

```
from time import time
import copy
class Obj:
    def __init__(self, created=time()):
        self.created = created
        print "init", created
    def __getinitargs__(self):
        print "getinitargs", self.created
        return (self.created,)
    def __getstate__(self):
        print "getstate", self.created
        return {'created':self.created}
    def __setstate__(self, dict):
        print "setstate", dict
        self.mynum = dict['created']
    def __repr__(self):
        return "created "+str(self.created)

a = Obj()
print a
b = copy.deepcopy(a)
print b
```

Результат выполнения:

```
init 981109784.654
created 981109784.654
getinitargs 981109784.654
init 981109784.654
getstate 981109784.654
setstate {'created': 981109784.65362799}
created 981109784.654
```

В данном примере при копировании объекта сохраняется время его создания. По выводимым значениям можно проследить порядок вызова функций.

Примечание

Начиная с версии 2.2, описанные здесь приемы для управления копированием могут измениться.

9.2.4. Модуль *gc*

Модуль `gc` является интерфейсом к механизму сборки мусора интерпретатора Python. Python решает проблемы освобождения памяти при помощи подсчета ссылок *(обсуждение этого механизма см. в разд 8.2)*. Однако в случае циклических ссылок объекты могут остаться в памяти в виде мусора. Модуль `gc` доступен только в том случае, если интерпретатор Python был скомпилирован со сборщиком мусора. В противном случае при импорте модуля будет возбуждено исключение `ImportError`.

Сборщик мусора играет вспомогательную роль в интерпретаторе: предполагается, что программы не должны оставлять структуры данных с циклическими ссылками, и может быть отключен, если есть уверенность, что программа не накапливает в памяти мусор.

Следующий пример показывает самый простой случай появления циклических ссылок и работу сборщика мусора:

```
l = [None, 1, 2, 3]  # заводим список
l[0] = l    # создаем циклическую ссылку из элемента списка на список
del l       # удаляем список (в памяти он остался!)
import gc
gc.set_debug(gc.DEBUG_LEAK)  # включаем режим отладки
print "Собрано объектов:", gc.collect()
print "Наш мусор:", gc.garbage
```

Результат выполнения:

```
gc: collectable <list 0x40152e4c>
Собрано объектов: 1
Наш мусор: [[[...], 1, 2, 3]]
```

Предупреждение

В приведенном примере память из-под мусора не освобождается, т. к. список `gc.garbage` ссылается на собранные объекты. Такое поведение задается флагом `gc.DEBUG_LEAK`.

В табл. 9.3 рассмотрены функции и переменные модуля `gc`.

Таблица 9.3. *Функции и переменные модуля* `gc`

Функция	Действие
`gc.enable()`	Включает автоматическую сборку мусора
`gc.disable()`	Выключает автоматическую сборку мусора
`gc.isenabled()`	Спрашивает, включена ли автоматическая сборка мусора

Таблица 9.3 *(окончание)*

Функция	Действие
`gc.collect()`	Производит сборку мусора среди всех поколений объектов. Функция возвращает количество собранных объектов
`gc.set_debug(flag)`	Устанавливает режим отладки согласно заданным флагам
`gc.get_debug()`	Получает информацию о текущем режиме отладки
`gc.get_threshold()`	Получает текущие пороги автоматического включения сборки мусора
`gc.set_threshold (t0[, t1[, t2]])()`	Устанавливает пороги автоматического включения сборки мусора (то есть задает частоту сборки мусора)
`gc.garbage`	Список объектов, которые являются мусором и не могут быть освобождены автоматически. Отладочный флаг `gc.SAVEALL` влияет на то, какие объекты будут записываться в этот список

Флаги задания режима отладки приведены в табл. 9.4. Объекты, которые находит сборщик мусора, могут быть освобождаемыми (collectable) или неосвобождаемыми (uncollectable), могут быть собственно объектами (objects) или объектами, которые являются экземплярами класса (instances). Флаги могут комбинироваться с помощью побитового ИЛИ ("`|`").

Таблица 9.4. *Флаги отладки модуля* `gc`

Флаг	Если флаг установлен
`gc.DEBUG_COLLECTABLE`	Печатать информацию об освобождаемых объектах
`gc.DEBUG_UNCOLLECTABLE`	То же, об объектах, которые не могут быть освобождены автоматически
`gc.DEBUG_INSTANCES`	Печатать данные об экземплярах классов
`gc.DEBUG_OBJECTS`	Печатать данные о собственно объектах
`gc.DEBUG_SAVEALL`	Сохранять все найденные объекты в `garbage` вместо того, чтобы освобождать из-под них память
`gc.DEBUG_LEAK`	Набор всех вышеперечисленных флагов, позволяющий отлаживать программу на предмет утечки памяти
`gc.DEBUG_STATS`	Печатать статистику по сборке мусора

Из всех перечисленных функций, переменных и констант, пожалуй, только `gc.set_threshold()` требует дополнительного объяснения. Согласно документации, сборщик мусора классифицирует объекты по трем поколениям. В самое молодое, нулевое, поколение попадают все новые объекты. Объект,

который не был собран процедурой сборки мусора, попадает в следующее поколение, либо остается в самом старшем, втором, поколении. Для принятия решения о сборке мусора сборщик ведет учет числа размещений и освобождений памяти с момента последней сборки мусора. Когда разница превысит `t0`, вызывается процедура сборки. При этом проверяется только нулевое поколение. Поколение 1 проверяется только после того, как поколение 0 было проверено в `t1` процедурах сборки мусора. Аналогично, поколение 2 будет проверено только после `t2` проверок поколения 1.

9.2.5. Модуль *getpass*

Модуль `getpass` предоставляет две небольшие, но полезные функции.

- `getpass.getpass([prompt]) -> s`

  Предлагает пользователю ввести пароль. Необязательным аргументом `prompt` можно задать строку подсказки (по умолчанию `"Password: "`). Возвращает строку с паролем.

- `getpass.getuser() -> s`

  Возвращает *входное имя (login)* пользователя. Функция доступна на платформах UNIX и Windows.

9.2.6. Модуль *operator*

Модуль `operator` предоставляет интерфейс к встроенным операторам и операциям Python в виде обычных функций (табл. 9.5). Для удобства, имена приведены как с двойными подчеркиваниями, так и без них. (Следует сразу заметить, что варианты имен с подчеркиваниями не импортируются оператором `from-import` по умолчанию). К представлению операций как функций приходится прибегать в случаях, когда нам нужно передать некоторую встроенную функцию в качестве аргумента другой функции, например, функциям `map()` или `reduce()`. Кроме того, некоторые операции осуществляются в Python неявно. Например, при необходимости определить истинностное значение некоторого выражения можно использовать функцию `operator.truth()`, а узнать, является ли некоторый объект последовательностью, можно с помощью `operator.isSequenceType()`.

```
>>> import operator
>>> from operator import *
>>> print "S =", reduce(add, [1,2,3,4,5], 0)
S = 15
>>> print "Сменили знак:", map(neg, [1,2,3])
Сменили знак: [-1, -2, -3]
>>> print "A 10 раз: ", repeat("A",10)
```

```
А 10 раз:  AAAAAAAAAA
>>> print "Истинностное значение []:", truth([])
Истинностное значение []: 0
>>> print "not []:", operator.__not__([])
not []: 1
```

Таблица 9.5. *Соответствие операций и функций*

Операция	Функция модуля `operator`
`x + y`	`__add__(x, y), add(x, y)`
`x & y`	`and_(x, y)`
`x and y`	`__and__(x, y)`
`x + y`	`concat(x, y), __concat__(x, y)`
`y in x`	`contains(x, y)`
`del x[i]`	`delitem(x, i), __delitem__(x, i)`
`del x[i:k]`	`delslice(x, i, k), __delslice__(x, i, k)`
`x / y`	`div(x, y, __div__(x, y)`
`x[i]`	`getitem(x, y), __getitem__(x, y)()`
`x[i:j]`	`getslice(x, i, j, __getslice__(x, i, j)`
`~x`	`inv(x), __inv__(x)`
`x << y`	`lshift(x, y), __lshift__(x)`
`x % y`	`mod(x, y), __mod__(x, y)`
`x * y`	`mul(x, y), __mul__(x, y)`
`-x`	`neg(x), __neg__(x)`
`not x`	`not_(x), __not__(x)`
`x | y`	`or_(x, y)`
`x or y`	`__or__(x, y)`
`+x`	`pos(x), __pos__(x)`
`seq * i`	`repeat(seq, i, __repeat__(seq, i)`
`x >> y`	`rshift(x, y), __rshift__(x, y)`
`x[i] = y`	`setitem(x, i, y), __setitem__(x, i, y)`
`x[i:j] = y`	`setslice(x, i, j, y), __setslice__(x, i, j, y)`
`x - y`	`sub(x, y), __sub__(x, y)`
`x ^ y`	`xor(x, y), __xor__(x, y)`

Кроме того, в модуле определены следующие функции:

- `operator.indexOf(x, y)` — возвращает первый индекс, по которому `y` находится в `x` (или возбуждает `ValueError`);
- `operator.countOf(x, y)` — подсчитывает, сколько раз `y` встречается в `x`;
- `operator.truth(x)` — возвращает истинностное значение объекта `x`, т. е. `0` или `1`.

В следующем примере мы построим таблицу свойств различных типов, из которых будет ясно действие функций `isCallable()`, `isMappingType()`, `isNumberType()` и `isSequenceType()`.

Листинг 9.11. Строим таблицу свойств различных типов

```
class eg:
  def __init__(self): pass
  def __repr__(self): return "объект класса " + self.__class__.__name__
class eg1(eg):
  def __call__(self, x): return x*2
def f(x): return x

from operator import *
import operator
checks = [
          isNumberType,           # (N) - числовой тип
          isSequenceType,         # (S) - тип последовательность
          isMappingType,          # (M) - тип словарь
          isCallable,             # (C) - объекты данного типа
                                  #  можно вызывать на исполнение
          ]
examples = [None, 1, 1.2, "abc", [], {1:'1',2:'2'},
            abs, f, eg, eg.__init__, eg(), eg1(), operator]
print " "*32, ":", "N S M C"      # выводим шапку таблицы
for x in examples:                # для всех примеров
  print "%32s"%`x`[:32], ":",       # выводим название объекта
  for chk in checks:                # для всех проверок
    print ["-","+"][apply(chk,[x])],  # выводим результат проверки
  print
```

Результат выполнения:

```
                                  : N S M C
                               None : - - - -
                                  1 : + - - -
                                1.2 : + - - -
                              'abc' : - + - -
                                 [] : - + - -
                  {2: '2', 1: '1'} : - - + -
          <built-in function abs> : - - - +
        <function f at 0x80dba14> : - - - +
<class __main__.eg at 0x80db82c> : - - - +
    <unbound method eg.__init__> : - - - +
                 объект класса eg : + + + -
                объект класса eg1 : + + + +
<module 'operator' from '/usr/lo : - - - -
```

В приведенном примере мы применяем функции для проверки типа к объектам различных типов. Видно, что любой пользовательский объект формально является числом, последовательностью и словарем, но только объекты, имеющие метод `__call__()`, могут вызываться на исполнение. (В примере класс `eg1` отличается от `eg` только методом `__call__()`).

9.2.7. Модули *pickle* и *cPickle*

Модули `pickle` и `cPickle` обеспечивают преобразование (почти произвольных) объектов Python в последовательность байтов и обратно (консервировать и расконсервировать). Указанное преобразование называется *сериализацией объектов (object serialization)*. Сериализованные ("консервированные") объекты легко сохранять в файле на диске, в базе данных или передавать по сети. При восстановлении сериализованных объектов классы, к которым они относятся, должны быть определены, т. к. `pickle` сохраняет только данные, но не всю структуру класса.

Могут быть сериализованы следующие объекты:

- встроенные типы: `None`, числа, строки (обычные и Unicode);
- списки, кортежи и словари, содержащие только сериализуемые объекты;
- функции, определенные на уровне модуля (сохраняется имя, а не реализация!);
- встроенные функции;
- классы, определенные на уровне модуля;

- объекты классов, `__dict__()` или `__setstate__()` которых являются сериализуемыми.

Пример в следующих в двух листингах (отправителе и получателе) проиллюстрирует основную задачу `pickle`. Для удобства изложения данные передаются посредством файла на диске.

Листинг 9.12. Отправитель объекта

```
import pickle, time
output_file = open("object.dat", "w")
p = pickle.Pickler(output_file)
p.dump("Сохраненное время:")
p.dump(time.localtime(time.time()))
output_file.close()
```

Листинг 9.13. Получатель объекта

```
import pickle
input_file = open("object.dat", "r")
print pickle.load(input_file),
print time.strftime("%d.%m.%Y %H:%M", pickle.load(input_file))
input_file.close()
```

Результат выполнения:

```
Сохраненное время: 30.06.2001 15:51
```

С помощью метода `dump()` объекта класса `Pickle` мы записываем объект (в нашем случае это кортеж из девяти элементов) в открытый файл. Второй листинг предлагает использование функции `load()` для восстановления сериализованного объекта из файла. Мы могли бы воспользоваться объектом `UnPickler` по аналогии с тем, как это делается в программе на первом листинге.

Таблица 9.6. *Функции модулей* `pickle` *и* `cPickle`

Функция (или конструктор)	Назначение
`Pickler(file [, bin]) -> p`	Создает Pickler-объект для записи сериализаций объектов в файл file. Если необязательный параметр `bin` не ноль, запись производится в бинарном формате. В противном случае используется текстовый формат
`UnPickler(file) -> unp`	Создает UnPickler-объект для чтения сериализаций из файла file

Таблица 9.6 *(окончание)*

Функция (или конструктор)	Назначение
`dump(obj, file[, bin])`	Записывает сериализованный объект `obj` в открытый файл file. Про `bin` см. выше
`dumps(obj [, bin]) -> s`	То же, но возвращает строку с последовательностью байтов
`load(file) -> obj`	Восстанавливает очередной объект из файла file
`loads(s) -> obj`	То же, но из строки `s`

Модуль `cPickle` является намного более эффективной реализацией `pickle`. Его отличие состоит в том, что он не предоставляет классов `Pickler` и `UnPickler`, от которых можно наследовать. В `cPickle` присутствуют одноименные функции для порождения соответствующих объектов. В большинстве случаев использование `cPickle` предпочтительнее.

В случае возникновения ошибок при сериализации (и восстановлении) возбуждается исключение `PicklingError` (`UnPicklingError`).

Описываемые здесь модули используются в модуле `shelve` *(см. разд. 9.2.10)*.

Замечание

Сериализация и последующее восстановление объектов сравнительно безопасная процедура, т. к. реализация функций и методов при этом не передается. Для более низкоуровневой сериализации объектов встроенных типов можно использовать стандартный модуль `marshal`, не описанный в этой книге.

9.2.8. Модуль *pprint*

Модуль `pprint` (от англ. *pretty print* — красивая печать) позволяет более удобно печатать списки, кортежи и словари. Модуль предоставляет несколько функций и класс `PrettyPrinter`, с помощью которого можно более точно настраивать параметры вывода.

Функция `pprint.pformat()` создает строку с красивым представлением объекта, заданного аргументом. Функция `pprint.pprint()` печатает объект (первый аргумент) в открытый поток (второй аргумент) или на стандартный вывод (если второй аргумент не задан). Следующий пример иллюстрирует работу этих функций:

```
>>> import pprint
>>> l = [[0]*5]*5
>>> print l
```

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
>>> pprint.pprint(l)
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
import pprint
print pprint.pformat([range(0,1000,250)]*2+[[-10]*4]*2)
[[0, 250, 500, 750],
 [0, 250, 500, 750],
 [-10, -10, -10, -10],
 [-10, -10, -10, -10]]
```

Класс `PrettyPrinter` дает возможность задавать некоторые значения при вызове конструктора.

```
PrettyPrinter(indent=1, width=80, depth=None, stream=None) -> p
```

где `indent` — количество пробелов для отступа при переходе на следующий уровень вложенности; `width` — ширина вывода; `depth` — максимальная глубина представления вложенных объектов; `stream` — поток для вывода (по умолчанию — стандартный поток вывода).

Соответственно, объекты класса `PrettyPrinter` имеют методы `pprint()` и `pformat()`, которые аналогичны функциям, описанным выше.

Модуль `PrettyPrinter` содержит и другие полезные функции, с которыми лучше познакомиться по документации к Python или в интерактивном сеансе работы с Python.

9.2.9. Модуль *repr*

Модуль `repr` предоставляет возможности, схожие с возможностями встроенной функции `repr()`, но позволяет ограничивать размер выводимой информации, что очень удобно при отладке. Следующий пример показывает возможности `repr`:

```
import repr
print repr.repr("*"*300)
print repr.repr(range(300))
repr.aRepr.maxstring = 70
print repr.repr("*"*300)
```

Результат выполнения:

```
'************...************'
[0, 1, 2, 3, 4, 5, ...]
'******************************...******************************'
```

Модуль `repr` предоставляет класс `Repr`, экземпляр этого класса `aRepr` и функцию `repr()`, которая на самом деле является методом `aRepr.repr()`, и потому на нее влияют значения атрибутов объекта `aRepr`. Набор атрибутов `aRepr` и их значения даны ниже:

```
from repr import aRepr, repr
aRepr.maxlevel = 6     # макс. уровень вложенности
aRepr.maxtuple = 6     # макс. длина представления кортежа
aRepr.maxlist = 6      # " " " списка
aRepr.maxdict = 4      # " " " словаря
aRepr.maxstring = 30   # " " " строки
aRepr.maxlong = 40     # " " " длинного целого
aRepr.maxother = 20    # " " " других объектов
```

От класса `repr.Repr` можно наследовать свои классы. Пример ниже показывает, как можно добавить число поддерживаемых типов, а также как заменить метод представления строки, чтобы он печатал русские буквы, а не коды символов:

Листинг 9.14. Пример модификации класса `repr.Repr`

```
import repr, sys, string
class CyrRepr(repr.Repr):
    def repr_file(self, obj, level):  # взят из документации к Python
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return `obj`
    def repr_string(self, x, level):
        def cyr_repr(s):
          return "'"+string.join(map(
              lambda x: ord(x) > 32 and x or `x`[1:-1],
              s), "")+"'"
        s = cyr_repr(x[:self.maxstring])
        if len(s) > self.maxstring:
            i = max(0, (self.maxstring-3)/2)
            j = max(0, self.maxstring-3-i)
```

```
            s = cyr_repr(x[:i] + x[len(x)-j:])
            s = s[:i] + '...' + s[len(s)-j:]
        return s

aRepr = CyrRepr()
print aRepr.repr(sys.stdin)
print repr.aRepr.repr("пример\n"*10)
print aRepr.repr("пример\n"*10)
```

Результат выполнения:

```
<stdin>
'\320\322\311...5\305\322\012'
'пример\012пр...012пример\012'
```

Замечание

Те, кто интересуются, как для объекта класса файл был вызван обработчик `repr_file()`, могут обратиться к исходному тексту модуля `repr`.

9.2.10. Модуль *shelve*

Модуль `shelve` позволяет сохранять (почти) произвольные объекты Python во внешнем файле. Shelf (полка) — *устойчивый (persistent) объект*, напоминающий словарь. В отличие от `dbm` *(см. разд. 9.8.1)*, значения, хранимые на полке, могут быть любыми объектами, которые способен обрабатывать модуль `pickle` *(см. разд. 9.2.7)*. Ключами являются строки. Полка хранится при помощи простейшей базы данных, одной из разновидностей `dbm`. Это могут быть `gdbm`, `dbm` или что-то другое, в зависимости от того, что доступно в системе. В некоторых реализациях может требоваться явный вызов метода `sync()`. Одновременное чтение полки несколькими процессами всегда является безопасной операцией, но для обеспечения безопасного одновременного доступа для (множественного) чтения/записи требуется блокировать файл для записи. В UNIX это можно сделать при помощи механизма lock-файлов *(см. разд. 9.9.2)*.

Интерфейс этого модуля лучше рассмотреть на прокомментированном примере в листинге 9.15.

Листинг 9.15. Файл myshelve.py. Демонстрация работы с полкой

```
import shelve
data = [1,2,3,4]          # -- данные (объект)
key = "dat"               # -- ключ (строка)
```

```
filename = "myshl.dat"  # -- имя файла для хранения полки
d = shelve.open(filename)       # открытие полки
d[key] = data           # сохранить данные под ключом key
                        # (затирает старое значение, если оно было)
data = d[key]           # загрузить значение по ключу
len(d)                  # получить количество объектов на полке
d.sync()                # запись изменений в БД на диске
del d[key]              # удалить ключ и значение
flag = d.has_key(key)   # проверка наличия ключа
list = d.keys()         # список ключей
d.close()               # закрытие полки
```

При работе могут возбуждаться исключения: `KeyError` — в случае отсутствия ключа, `anydbm.error` — если открываемый файл не содержит базы данных *(см. разд. 9.8.1)*, `cPickle.UnpickleableError` — в случае невозможности поместить некоторый объект на полку.

9.2.11. Модуль *traceback*

Модуль `traceback` оказывает неоценимую услугу при отладке программ, особенно в таких "труднодоступных" местах, как CGI-сценарии. Этот модуль извлекает, форматирует и выводит трассировочную информацию из трассировочных объектов *(см. разд. 8.2)*. Обычно необходимость в этом появляется при отладке программ и связанной с ней обработке исключений.

В модуле определены следующие функции.

- `traceback.print_tb(tb[, limit[, file]])`

  Печатает до `limit` (по умолчанию — все) позиций стека исполнения трассировочного объекта `tb` в файлоподобный объект `file` (по умолчанию `sys.stderr`).

- `traceback.print_exception(exc_type, exc_value, tb[, limit[, file]])`

  Печатает информацию об исключении типа `exc_type` со значением `exc_value` и трассировочным объектом `tb`. Параметры `limit` и `file` имеют те же значения, что и у предыдущей функции.

- `traceback.print_exc([limit[, file]])`

  Укороченная запись вызова `traceback.print_exception()`, где в качестве первых трех аргументов берутся значения функции `sys.exc_info()`.

- `traceback.print_last([limit[, file]])`

  Укороченная запись вызова `traceback.print_exception()`, где в качестве первых трех аргументов берутся значения `sys.last_type`, `sys.last_value` и `sys.last_traceback`, оставшиеся после последнего исключения.

- `traceback.print_stack([f[, limit[, file]]])`

  Печатает данные фреймового объекта `f` (или текущего фрейма исполнения, если `f` не задан). Параметры `limit` и `file` имеют те же значения, что и ранее.

- `traceback.extract_tb(tb[, limit]) -> list`

  Возвращает список из не более чем `limit` элементов (если `limit` задан), в которых записаны кортежи вида `(filename, lineno, funcname, text)` с трассировочной информацией. Здесь `filename` и `lineno` — имя файла и номер строки в этом файле, `funcname` — имя функции, а `text` — текст строки исходного кода (или `None`, если он недоступен).

- `traceback.extract_stack([f[, limit]]) ->`

  Возвращает список данных из фреймового объекта `f` в том же формате, что и функция `traceback.extract_tb()`.

- `traceback.format_list(list) -> list1`

  Возвращает список строк, отформатированных на основе списка кортежей, какие возвращают две предыдущие функции.

- `traceback.format_exception_only (exc_type, exc_value) -> list`

  Форматирует часть трассировки, соответствующую исключению `exc_type`, `exc_value`, возвращая список готовых к печати строк.

- `traceback.format_exception(exc_type, exc_value, tb[, limit]) -> list`

  Форматирует данные об исключении, возвращая список строк, готовых к выводу. Результат объединения этих строк получается тем же, что и при использовании функции `traceback.print_exception()`.

- `traceback.format_tb(tb[, limit]) -> list`

  Аналогична `{traceback.format_list(extract_tb(tb, limit))}`.

- `traceback.format_stack ([f[, limit]]) -> list`

  Аналогична `format_list(extract_stack(f, limit))`.

- `traceback.tb_lineno(tb) -> i`

  Возвращает номер строки, записанный в объекте `tb`. Аналогична `tb.tb_lineno`, но возвращает правильный результат в условиях применения оптимизации

Полезный пример работы с трассировочным объектом приведен в листинге 9.16.

Листинг 9.16. Файл trace_example.py. Использование трассировочной информации для отладки

```
def dbg_except():
  import traceback, sys, string
```

```
  """Функция для отладки операторов try-except"""
  import string
  print "-"*70
  print string.join(apply(traceback.format_exception, sys.exc_info()))
  print "-"*70

def bad_function(x, y):
  """Опасная функция"""
  return x/y

try:
  bad_function(0, 0)
except:
  dbg_except()
```

Результат выполнения:

```
----------------------------------------------------------------------
Traceback (most recent call last):
   File "trace_example.py", line 14, in ?
    bad_function(0, 0)
   File "trace_example.py", line 11, in bad_function
    return x/y
 ZeroDivisionError: integer division or modulo by zero

----------------------------------------------------------------------
```

9.2.12. Модуль *types*

Для правильного использования встроенной функции `type()` модуль `types` предоставляет материал для сравнений, давая объекты-типы для всех встроенных типов. В табл. 9.7 приведены все константы, определяемые в модуле `types`.

Таблица 9.7. *Константы модуля* `types`

Название константы	Обозначаемый тип
`BufferType`	Буфер
`BuiltinFunctionType`	Встроенная функция
`BuiltinMethodType`	Встроенный метод
`ClassType`	Класс

Таблица 9.7 *(окончание)*

Название константы	Обозначаемый тип
`CodeType`	Кодовый объект
`ComplexType`	Комплексное число
`DictType`, `DictionaryType`	Словарь
`EllipsisType`	Тип `Ellipsis`
`FileType`	Файл
`FloatType`	Число с плавающей точкой
`FrameType`	Фрейм исполнения
`FunctionType`	Функция
`InstanceType`	Экземпляр класса
`IntType`	Целое число
`LambdaType`	Функция, определенная лямбда-выражением
`ListType`	Список
`LongType`	Целое произвольной точности
`MethodType`	Метод
`ModuleType`	Модуль
`NoneType`	Тип `None`
`SliceType`	Срез
`StringType`	Строка
`TracebackType`	Трассировочный объект
`TupleType`	Кортеж
`TypeType`	Тип тип
`UnboundMethodType`	Несвязанный метод
`XRangeType`	Тип диапазон

Следующий пример показывает, что при желании в Python можно легко организовать проверку типов аргументов функции:

```
from types import *
def typed_func(x, y, z):
  assert map(type, (x, y, z)) == [IntType, IntType, ListType],\
         "непредусмотренные типы"
  return x+y+len(z)
```

```
print typed_func(1, 2, [1, 2, 3])
print typed_func(1, 2, 2)
```

Результат выполнения:

```
6
Traceback (innermost last):
  File "<stdin>", line 8, in ?
  File "<stdin>", line 3, in typed_func
AssertionError: непредусмотренные типы
```

9.2.13. Модуль *UserString*

Хотя в Python все значения являются объектами, не все из них — экземпляры классов, а потому от встроенных типов нельзя наследовать другие типы объектов. Модуль `UserString`, появившийся в Python 2.0, призван обеспечить определение класса `UserString`, объекты которого по своему поведению почти неотличимы от встроенных строк языка Python, а от класса `UserString`, разумеется, можно наследовать другие классы. В этом же модуле дан пример такого наследования — класс `MutableString` (изменяемая строка). Этот класс дополняет обычные строки методами, позволяющими изменять строку на месте. В следующем примере мы видим пример использования классов модуля `UserString`:

```
from UserString import *
s1 = UserString("строка1")
s2 = MutableString("строка2")
print s1 + s2
s2[0:3] = "СТР"
print s2
```

Результат выполнения:

```
строка1строка2
СТРока2
```

Разумеется, использование написанных на Python классов для строк не может не сказываться на производительности, поэтому классы из данного модуля в основном носят иллюстративный характер. Их применение ограничено теми случаями, когда на базе класса `UserString` совершенно необходимо построить другой класс, очень похожий по своему поведению на строки.

9.2.14. Модули *UserList* и *UserDict*

Модуль `UserList` предоставляет программисту класс `UserList`, объекты которого ведут себя аналогично спискам. Нетрудно догадаться, что модуль `UserDict` делает то же самое для словарей.

Как и в случае с `UserString`, необходимость в классах `UserList` и `UserDict` возникает в случае, когда некоторый класс очень похож на список (или словарь), но помимо методов, присущих спискам (словарям), должен иметь и некоторые дополнительные методы. Конечно, встроенные списки и словари языка Python работают быстрее их собратьев, порождаемых на основе классов `UserList` и `UserDict`, но, как это часто случается в Python, побеждает более выразительное и краткое решение. В следующем примере мы расширяем класс `UserDict`, внеся в него возможность получать отсортированные (по ключам) списки ключей и значений:

```
import UserDict

class SortableDict(UserDict.UserDict):
  def sorted_keys(self):
    tmplist = self.keys()
    tmplist.sort()
    return tmplist
  def sorted_items(self):
    tmplist = self.items()
    tmplist.sort()
    return tmplist
  def sorted_values(self):
    tmplist = self.items()
    tmplist.sort()
    return map(lambda x: x[1], tmplist)

dict = SortableDict({'a': 3, 'b': 2, 'c': 1})

print dict.items()
print dict.sorted_items()
print dict.sorted_values()
```

Результат выполнения:

```
[('b', 2), ('c', 1), ('a', 3)]
[('a', 3), ('b', 2), ('c', 1)]
[3, 2, 1]
```

9.2.15. Модули *__builtin__* и *__main__*

Модуль `__builtin__` — это модуль, содержащий встроенные функции. Например, встроенная функция `len()` имеет полное имя `__builtin__.len()`.

Модуль `__main__` — этот главный модуль исполняющейся программы. Самое главное применение `__main__` состоит в том, что можно проверять, исполняется ли некоторый модуль как главный модуль (в этом случае можно запустить некоторую демонстрационную или тестовую функцию) или нет:

```
# В конце модуля:
if __name__ == "__main__":
  demo()
```

Например, модуль `mimify` можно как импортировать в программу, так и исполнять как самостоятельную программу.

9.3. Обработка строк

Одним из важных применений Python является обработка текстовых данных. В этом разделе вы найдете модули, работающие со строками. Стоит заметить, что почти все функции модуля `string` в Python версии 2.0 и старше перенесены в методы объектов-строк.

9.3.1. Модуль *string*

Модуль `string`, наверное, один из самых используемых в Python модулей. С появлением в Python 2 .0 методов у строк, во многих случаях отпадает необходимость его импортировать, т. к. нужные функции нашли отражение в методах. В силу указанного обстоятельства, ниже будут рассмотрены только те функции, которые есть в Python 1.5.2. Тем, кто использует более поздние версии Python, можно порекомендовать методы строк *(см. статью "Строки" в разд. 8.2.3)*.

В табл. 9.8 приведены функции модуля `string`, а в табл. 9.9 — его константы.

Таблица 9.8. *Функции модуля* `string`

Функция	Описание
`string.atof()`, `string.atoi()`, `string.atol()`	Устаревшие функции. Пользуйтесь встроенными функциями `float()`, `int()`, `long()`
`string.capitalize(s) -> s1`	Сделать первую букву прописной
`string.capwords(s) -> s1`	Сделать первые буквы всех слов прописными

Таблица 9.8 *(продолжение)*

Функция	Описание
`string.center(s, w) -> s1`	Отцентрировать строку `s` в поле длины `w`
`string.count(s, sub [, start[, end]]) -> i`	Число вхождений строки `sub` в строке `s[start:end]`
`string.expandtabs (s[, tabsize]) -> s1`	Замена табуляций на пробелы. По умолчанию величина табуляции `tabsize` равна `8`
`string.find(s, sub [,start [,end]]) -> i`	Возвращает наименьший индекс, с которого начинается вхождение подстроки `sub` в строку `s`. Как обычно, `start` и `end` ограничивают поиск окном `start:end`, но индекс выдается по исходной строке. Если подстрока не найдена, возвращается `-1`
`string.index(s, sub [,start [,end]]) -> i`	Аналогично `find()`, но возбуждает `ValueError` в случае неудачи
`string.join(seq, s) -> s1, string.joinfields(seq, s) -> s1`	Соединение строк из `seq` через разделитель `s`
`string.ljust(s, w) -> s1`	Возвращает строку, выровненную по левому краю с помощью пробелов в поле ширины `w`
`string.lower(s) -> s1`	Приводит строку к нижнему регистру букв
`string.lstrip(s) -> s1`	Удаляет пробельные символы слева
`string.maketrans(old, new) -> s`	Создает таблицу перекодировки символов, которую затем можно использовать в функции `translate()`. Строки `old` и `new` задают соответствие символов и должны быть одинаковой длины
`string.replace (s, old, new[, n]) -> s1`	Возвращает копию строки `s`, в которой подстроки `old` заменены на `new`. Если задано `n` — то заменяются только первые `n` вхождений
`string.rfind(s, sub [, start [,end]]) -> i`	Возвращает наибольший индекс, начиная с которого строка `sub` располагается в строке `s`. Если задан `start`, то поиск происходит в `s[start:]`, а если есть и `end`, то в `s[start:end]`. Индекс все равно соответствует исходной строке. Если подстрока в строке не найдена, возвращается `-1`
`string.rindex(s, sub [, start [,end]]) -> i`	Аналогично `rfind()`, но в случае неудачи возбуждается исключение `ValueError`
`string.rjust(s, w) -> s1`	Выравнивание по правому краю в поле шириной `w`. При необходимости строка дополняется слева пробелами

Таблица 9.8 *(окончание)*

Функция	Описание
`string.rstrip(s) -> s1`	Удаляет пробельные символы справа
`string.split(s[, sep [, n]]) -> list, string.splitfields(s [, sep[, n]]) -> list`	Возвращает список подстрок, получающихся разбиением строки `a` разделителем `sep`. Параметр `n` задает максимальное количество разбиений
`string.strip(s) -> s1`	Удаляет пробельные символы с обоих краев
`string.swapcase(s) -> s1`	Меняет регистр букв на обратный
`string.translate(s, table[, deletechars]) -> s1`	Производит преобразование с помощью таблицы перекодировки `table`. Аргумент `table` задает строку перекодировки на 256 символов (ее можно сделать с помощью функции `string.maketrans()`). Необязательный параметр `deletechars` задает строку с символами, которые подлежат удалению
`string.upper(s) -> s1`	Переводит буквы строки в верхний регистр
`string.zfill(arg, w) -> s`	Дополнение строки с числом (`arg`) нулями слева до ширины `w`

Таблица 9.9. *Константы модуля* `string`

Константа	Описание
`string.digits`	Цифры
`string.hexdigits`	Цифры шестнадцатеричной системы счисления
`string.letters`	Буквы (набор зависит от locale)
`string.lowercase`	Строчные буквы (набор зависит от locale)
`string.octdigits`	Цифры восьмеричной системы счисления
`string.uppercase`	Прописные буквы (набор зависит от locale)
`string.whitespace`	Пробельные символы

Далее будут рассмотрены несколько примеров применения наиболее употребительных функций:

```
import string
print string.replace("абракадабра", "абра", "ора"),
print string.replace("абракадабра", "абра", "ора", 1),
print string.replace("абракадабра", "абра", "аабра")
```

```
# В Python 2 можно и так:
print "абракадабра".replace("абра", "ора")
```

Результат выполнения:

```
оракадора оракадабра ааброкадаабра
оракадора
```

При использовании функции `string.replace()` не следует забывать, что *замены подстрок* производятся подряд. В следующем примере показано, как правильно произвести замену "при" на "пре" и наоборот с использованием функции `string.replace()`. Заметьте, что для того, чтобы пометить заменяемую подстроку при первой замене, использованы символы с кодами 1 и 2, поэтому наш вариант замены не приведет к проблемам только если в строке не используются символы с такими кодами (что верно для подавляющего большинства текстов).

```
from string import replace
eg = "не преемлем принебрежение прародой"
print replace(replace(replace(eg, "при", "\1при\2"),
                      "пре", "при"),
                      "\1при\2", "пре" )
print eg.replace("при","\1при\2")\
       .replace("пре","при")\
       .replace("\1при\2","пре")
```

Результат выполнения:

```
не приемлем пренебрежение природой
не приемлем пренебрежение природой
```

Следующий пример показывает применение функции `string.find()` для поиска:

```
>>> import string
>>> s = "one two three four five twenty one"
>>> print string.find(s, "one"), string.find(s, "two")
0 4
>>> print string.find(s, "one", 10)
31
>>> print string.find(s, "")
0
>>> print string.find(s, "six")
-1
```

С помощью функций `string.maketrans()` и `string.translate()` легко написать функцию для перекодировки текста (программа использует кодировку koi8-r):

Листинг 9.17. Функция для перекодировки из koi8-r в Windows-1251

```
import string

def koiwin(s,
           koiwin_tr=string.maketrans(
 'АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцчшщъыьэюя',
 'юабцдефгхийклмнопярстужвьызшэщчъЮАБЦДЕФГХИЙКЛМНОПЯРСТУЖВЬЫЗШЭЩЧЪ',),
           tr=string.translate):
  """koiwin_tr(koi) -> win"""
  return tr(s, koiwin_tr)

print koiwin("Вопрос")
```

Результат выполнения:

```
бНОПНЯ
```

О том, как делать перекодировку "правильно" в Python 2.0, есть пример в *разд. 9.3.4*.

В этой книге очень много примеров использования `string.split()` и `string.join()`, вот еще один, в котором IP-адрес переводится из точечной нотации в шестнадцатеричную:

```
from string import split, join
IP = "192.168.1.11"
print join(map(lambda x: "%02X"%int(x), split(IP, ".")), "")
print "%02X"*4 % tuple(map(int, split(IP, ".")))
```

Результат выполнения:

```
C0A8010B
C0A8010B
```

Сначала адрес разбивается функцией `split()` с использованием точки как разделителя, затем полученный список строк преобразуется функцией, определенной с помощью лямбда-выражения, после чего строки соединяются в одну строку с помощью `join()`. Второй вариант обходится без `join()`, т. к. IPv4-адреса имеют длину 4.

Кстати, функция `split()` может работать и без указания разделителя. В этом случае разделителем будет любое количество пробельных символов:

```
>>> from string import split, join
>>> s = "В этой строке   есть  лишние   пробелы\ти\табуляции."
>>> print join(split(s))
В этой строке есть лишние пробелы и табуляции.
```

Почти столь же часто применяется функция `string.strip()`. В следующем примере мы используем `strip()`, чтобы удалить лишние пробелы в текстовых полях таблицы:

```
from string import split, strip, join
table = """ Функция | Назначение
            split() | разбить строку на список строк
            join()  | соединить список строк в одну строку
            strip() | убрать пробелы по краям строки"""

for line in split(table, "\n")[1:]:
  print join(map(strip, split(line, "|")), " - ")
```

Результат выполнения:

```
split() - разбить строку на список строк
join() - соединить список строк в одну строку
strip() - убрать пробелы по краям строки
```

9.3.2. Модуль *re*

Регулярное выражение (regular expression) — специальный шаблон для поиска в строке, сопоставления со строкой, разбиения строки на части и других операций над строками. Одним регулярным выражением можно описать сразу много строк, поэтому они являются очень мощным средством для обработки текстовой информации, т. к. обладают большой описательной гибкостью.

По сравнению с обычной строкой, в шаблоне некоторые символы приобретают специальное значение, другие же (в основном буквы и цифры) в большинстве случаев обозначают самих себя.

Регулярные выражения в языках программирования Python и Perl, в утилитах grep, sed, awk используют похожие соглашения. Регулярные выражения, как и арифметические, имеют "операции", "операнды" и скобки и могут быть достаточно сложными. В регулярном выражении можно выделять группы символов, чтобы затем ссылаться на них как в объекте-результате

сравнения, так и далее, в том же регулярном выражении. В табл. 9.10 описываются символы, используемые в регулярных выражениях Python.

В этом разделе описаны регулярные выражения, доступные в модуле `re` стандартной библиотеки модулей Python.

Таблица 9.10. *Назначение символов в регулярных выражениях*

Символ	Назначение
`"."`	Соответствует любому символу (включая символ конца строки, если установлен флаг `DOTALL`)
`"^"`	Соответствует промежутку сразу после начала строки (любой из строк, если установлен флаг `MULTILINE`)
`"$"`	Соответствует промежутку перед концом строки (любой из строк, если установлен флаг `MULTILINE`)
`"*"`	Соответствует любому повторению или отсутствию предыдущего операнда (пытается производить сопоставление как можно большее число раз)
`"+"`	Соответствует любому повторению предыдущего операнда (пытается производить сопоставление как можно большее число раз)
`"?"`	Предыдущий операнд либо присутствует, либо отсутствует
`"{m,n}"`	Соответствует числу повторений предыдущего операнда от `m` до `n` включительно
`"*?"`, `"+?"`, `"??"`, `"{m,n}?"`	То же, что и `"*"`, `"+"`, `"?"`, `"{m,n}"`, но число соответствий сводится к минимуму
`"[...]"`	Соответствует любому символу из набора в скобках. (см. примеры)
`"[^...]"`	Не соответствует никакому символу из набора
`"\"`	Отменяет специальное значение следующего символа
`"\\"`	Соответствует одиночной косой черте `"\"`
`"|"`	Соответствует операнду справа или операнду слева (операция имеет низкий приоритет, поэтому операнды распространяются до ограничивающих скобок или до самых краев регулярного выражения)
`"(регвыр)"`	Соответствует регулярному выражению в скобках и выделяет группу
`"(?:регвыр)"`	То же, но не выделяет группы
`"(?=регвыр)"`	"Заглядывание" вперед: соответствует регулярному выражению, но как бы не занимает места в строке, к которой применяется шаблон. Таким образом, дальнейшее сопоставление с шаблоном начнется с того же места

Таблица 9.10 *(окончание)*

Символ	Назначение
"(?!регвыр)"	То же, но соответствует, если не соответствует регулярному выражению
"(?<=регвыр)"	Соответствует, если до этого момента соответствует регулярному выражению. Не занимает места в строке, к которой применяется шаблон. Параметр регвыр должен быть фиксированной длины (то есть, без "+" и "*")
"(?<!регвыр)"	То же, но с отрицанием
"(?P<имя>регвыр)"	Соответствует регулярному выражению и выделяет именованную группу с именем имя
"(?P=имя)"	Точно соответствует выделенной ранее именованной группе
"(?#...)"	Комментарий (игнорируется)
"(?флаг)"	Задает флаг для всего данного регулярного выражения. Флаги нужно задавать в начале шаблона

В табл. 9.11 описываются используемые в регулярных выражениях *специальные последовательности*.

Таблица 9.11. *Специальные последовательности*

Символ	Назначение
"\цифра"	Соответствует группе с номером, заданным цифрой. Группы нумеруются, начиная с 1
"\A"	Соответствует промежутку перед началом всей строки (почти аналогично "^")
"\Z"	Соответствует промежутку перед концом всей строки (почти аналогично "$")
"\b"	Соответствует промежутку между символами перед словом или после него
"\B"	Наоборот, не соответствует промежутку между символами на границе слова
"\d"	Цифра. Аналогично "[0-9]"
"\d"	Нецифра. Аналогично "[^0-9]"
"\s"	Любой пробельный символ. Аналогично "[\t\n\r\f\v]"
"\S"	Любой непробельный символ. Аналогично "[^\t\n\r\f\v]"
"\w"	Любая цифра или буква (зависит от флага LOCALE)
"\W"	Любая нецифра или небуква (зависит от флага LOCALE)

Некоторые функции для работы с регулярными выражениями принимают в качестве отдельного аргумента сочетание флагов, приведенных в табл. 9.12.

Таблица 9.12. *Флаги, используемые некоторыми функциями модуля* `re`

Флаг	Краткое имя	Полное имя	Описание
`i`	`re.I`	`re.IGNORECASE`	Сопоставление без учета регистра букв
`L`	`re.L`	`re.LOCALE`	Влияет на определение буквы в "`\w`", "`\W`", "`\b`", "`\B`" в зависимости от текущей "культурной среды" (locale)
`m`	`re.M`	`re.MULTILINE`	Если задан, "`^`" и "`$`" соответствуют началу и концу любой строки
`s`	`re.S`	`re.DOTALL`	Если задан, "`.`" соответствует также и символу конца строки "`\n`"
`x`	`re.X`	`re.VERBOSE`	Если задан, неэкранированные обратной косой чертой пробелы в шаблоне являются незначащими, а все, что после символа "`#`" — комментарием. Позволяет записывать регулярное выражение в несколько строк, улучшая его читаемость
`u`	`re.U`	`re.UNICODE`	В шаблоне и строке использован Unicode

В табл. 9.13 приведены примеры регулярных выражений.

Таблица 9.13. *Примеры регулярных выражений*

Пример	Комментарий
`"\b\w+\b"`	Соответствует слову
`"[+-]?\d+"`	Соответствует числу. Возможно, со знаком
`"\([+-]?\d+\)"`	То же, но стоящему в скобках. Так как скобки применяются в самих регулярных выражениях, их необходимо экранировать с помощью "`\`"
`"[a-cA-C]{2"}`	Соответствует строке из двух букв "`a`", "`b`" или "`c`". Например, "`Ac`", "`CC`", "`bc`"
`"aa|bb|cc|AA|BB|CC"`	То же, но буквы одинаковые
`"([a-cA-C])\1"`	То же, но с использованием групп
`"a(a|b)b"`	Соответствует "`aab`" или "`abb`"
`"^\d+:\s*(.*)$"`	Соответствует строке, которая начинается с числа и двоеточия. При этом все, что идет после двоеточия и, возможно, следующих за ним пробелов, выделяется в группу

Таблица 9.13 *(окончание)*

Пример	Комментарий
`"(\w+)=.*\b\1\b"`	Слово слева от "=" присутствует и справа. Используется сопоставление с группой по номеру (`\1`)
`"(?P<var>\w+)=.*\b (?P=var)\b"`	То же, но с помощью именованных групп
`"\bregular(?=\s+exp ression)"`	Соответствует слову `"regular"` только если за ним следует `"expression"`
`"(?<=regular ) expression"`	Соответствует слову `"expression"`, перед которым стоит `"regular"`

Модуль `re` содержит объекты `SRE_Pattern` и `SRE_Match`.

Для работы с регулярными выражениями можно использовать как функции, определенные в модуле `re` (они тоже описываются в этом разделе), так и методы объекта `SRE_Pattern`: выбор того или иного способа зависит от ситуации. Функции удобно применять для одиночного сопоставления строки с шаблоном, тогда как объекты больше подходят для массовых сопоставлений строк с одним и тем же шаблоном.

Объект `SRE_Pattern` получается в результате "компиляции" регулярного выражения функцией `re.compile()`. Компиляция готовит регулярное выражение к дальнейшему использованию. В табл. 9.14 приведены методы объекта класса `SRE_Pattern`.

Таблица 9.14. *Методы объектов класса* `SRE_Pattern`

Метод	Описание
`match(s[, pos [, endpos]]) -> mo`	Сопоставляет начало строки `s` с регулярным выражением. Возвращает `SRE_Match`-объект (`MatchObject` — в Python 1.*x*), если сопоставление удалось, и `None` при неудаче. Параметры `pos` (начальная позиция) и `endpos` (конечная позиция) позволяют задавать диапазон в исходной строке, в котором производится сопоставление
`search(s[, pos[, endpos]]) -> mo`	Аналогичен `match()`. Различие их состоит в том, что `match()` ищет соответствие шаблона и строки только в начале строки независимо от режима, задаваемого флагом `re.MULTILINE`
`split(s[, count] -> lst)`	Разбивает строку `s` по разделителям, которыми являются регулярные выражения. Параметр `count` указывает максимальное число разбиений (по умолчанию он равен нулю, что означает отсутствие ограничения). Если в регулярном выражении выделены группы, то список `lst` содержит не только разделенные компоненты, но и сами разделители (под каждую группу отводится отдельный элемент)

Таблица 9.14 *(окончание)*

Метод	Описание
`findall(s) -> lst`	Возвращает список строк, найденных в `s` по регулярному выражению в данном `SRE_Pattern`-объекте. Если шаблон содержал более одной группы, список содержит кортежи строк, где каждый элемент соответствует группе
`sub(repl, s[, count]) -> s1`	Заменяет в строке `s` вхождения подстрок, соответствующих регулярному выражению, на `repl`. Параметр `count` указывает максимальное число замен. Если `repl` является функцией, то она будет вызываться для каждой найденной подстроки. В качестве единственного параметра этой функции будет передаваться `SRE_Match`-объект. *(См. пример в листинге 3.6, разд. 3.4.)*
`subn(repl, s[, count]) -> (s1, num)`	Аналог `sub()`, возвращающий кортеж с результатом замен строкой `s1` и числом произведенных замен `num`

Помимо методов, `SRE_Pattern`-объекты имеют и другие атрибуты. Атрибут `flags` содержит флаги, использованные при компиляции регулярного выражения, `pattern` — исходное регулярное выражение, `groupindex` — словарь, устанавливающий соответствие между символическим названием группы и ее номером (равен `{}`, если именованные группы не были определены).

В модуле `re` определены функции, совпадающие именами с методами из табл. 9.14. Разумеется, в этих функциях в начале добавляется аргумент, соответствующий регулярному выражению, и иногда аргумент, задающий флаги. Ниже рассмотрены функции модуля `re`. Параметр `pat` может быть как строкой с регулярным выражением, так и компилированным регулярным выражением, т. е. `SRE_Pattern`-объектом.

```
re.compile(pat[, flags]) -> pobj
```

Компилирует в `SRE_Pattern`-объект регулярное выражение `pat`, принимая во внимание заданную комбинацию флагов `flags` (флаги комбинируются побитовым ИЛИ).

```
search(pat, s[, flags]) -> mobj
```

```
match(pat, s[, flags]) -> mobj
```

Эти две функции аналогичны методам `search()` и `match()`. Если сопоставление не состоялось, или в строке нет соответствующей шаблону подстроки, возвращается `None`. Следующий пример показывает разницу между `re.search()` и `re.match()` и влияние флага `re.MULTILINE` на интерпретацию "^" в регулярном выражении:

```
>>> import re
>>> print re.match("^ba", "good\nbad", re.M)
```

```
None
>>> print re.search("^ba", "good\nbad", re.M)
<SRE_Match object at 0x80d2790>
>>> print re.match("^ba", "good\nbad")
None
>>> print re.search("^ba", "good\nbad")
None
```

Выше было сказано, что "\A" и "\Z" аналогичны "^" и "$". Следующий пример демонстрирует это отличие (заметьте, что в регулярном выражении задан флаг re.M):

Результат выполнения:

```
>>> import re
>>> print re.findall("(?m)String\d$", "String1\nString2\n")
['String1', 'String2']
>>> print re.findall("(?m)String\d$", "String1\nString2")
['String1', 'String2']
>>> print re.findall("(?m)String\d\Z", "String1\nString2\n")
[]
>>> print re.findall("(?m)String\d\Z", "String1\nString2")
['String2']
```

Из примера видно, что "$" срабатывает в конце каждой строки, а "\Z" — соответствует только концу всей строки, не затрагивая и не принимая во внимание символ конца строки.

```
re.escape(s) -> s1
```

Функция добавляет обратную косую черту к некоторым символам строки s для подавления специального значения, которое они могли бы иметь в регулярном выражении без подобного экранирования. Функция обычно используется в случаях, когда строка должна стать частью регулярного выражения буквально, а мы заранее не знаем, какие символы в нее входят:

```
>>> import re
>>> s = "[*]"  # строка, которая требует экранирования
>>> r = re.escape(s)  # экранированная строка
>>> print r
\[\*\]
>>> # поиск:
>>> print re.findall("[ ]%s" % r, "a [*] b [*]")
[' [*]', ' [*]']
```

- `re.split(pat, s[, count]) -> lst`
- `re.findall(pat, s) -> lst`
- `re.sub(pat, repl, s[, count]) -> s1`
- `re.subn(pat, repl, s[, count]) -> (s1, num)`

Эти функции аналогичны одноименным методам `SRE_Pattern`-объектов. Заметьте, что для `flags` нет специального аргумента, поэтому флаги можно задать в начале регулярного выражения, например: `"(?i)(?L)регвыражение"`.

Следующий пример иллюстрирует работу функции `re.split`. Выделение группы в регулярном выражении разделителя приводит к включению его текста в список-результат:

```
>>> import re
>>> re.split(" ", "a b c")
['a', 'b', 'c']
>>> re.split("( )", "a b c")
['a', ' ', 'b', ' ', 'c']
>>> re.split("( -)", "a -b -c")
['a', ' -', 'b', ' -', 'c']
```

Результатом успешного сопоставления с шаблоном (регулярным выражением) является специальный объект — `SRE_Match` (или, в более старом Python — `MatchObject`). Этот объект содержит всю необходимую информацию о состоявшемся сопоставлении. Доступны следующие методы (табл. 9.15).

Таблица 9.15. *Методы объектов класса* `SRE_Match`

Метод	Описание
`start([g]) -> int`	Начало подходящего по шаблону фрагмента в исходной строке. Если задана группа `g`, то начало и конец конкретной группы
`end([g]) -> int`	Конец подходящего по шаблону фрагмента в исходной строке. Если задана группа `g`, то начало и конец конкретной группы
`span([g]) -> (start, end)`	Диапазон подходящего по шаблону фрагмента в исходной строке. Если задана группа `g`, то начало и конец конкретной группы
`expand(template) -> s`	Подставляет в строку `template` значения из нумерованных и именованных групп, найденных в результате сопоставления. То есть `"\1"`, ..., `"\9"` или `"\g<имя_группы>"` для именованных групп. Сочетания типа `"\n"` переводятся в соответствующие управляющие символы. Параметр `template` лучше всего задавать в виде сырой строковой константы, например: `r"\1 - \2\n"`

Таблица 9.15 *(окончание)*

Метод	Описание
`group([g1, g2 ..., gn]) -> tuple`	Возвращает кортеж из значений групп под номерами (или именами) `g1`, `g2`, ..., `gn`. Без аргументов возвращает группу `0` (весь подошедший к шаблону фрагмент). Если задана строка, то возвращается содержимое соответствующей ей именованной группы, если номер — содержимое группы с этим номером. Если группа не попала в соответствие, для соответствующего аргумента возвращается `None`. Для каждой группы берется ее последнее соответствие. При ошибках с именами (номерами) групп возбуждается исключение `IndexError`. Группы нумеруются в порядке следования открывающих скобок
`groupdict() -> dict`	Возвращает словарь именованных групп с их значениями. Если группа не попала в соответствие, ее значение равно `None`

Следующий пример демонстрирует работу с группами. В шаблоне выделено четыре группы, первая из которых имеет имя `num`. Символы, попавшие в группы, доступны с помощью метода `group()`.

```
>>> import re
>>> m = re.match(r"(?P<num>\d+)([a-z])?.\s*(\w+)\s+(\d)+",
...              "1.abc 12345 test")
>>> print m.group(0, 1, 2, 3, 4, "num")
('1.abc 12345', '1', None, 'abc', '5', '1')
```

В табл. 9.16 приведены примеры успешных и безуспешных сопоставлений. В левой колонке приведены аргументы функции `re.match()`, а в правой — результат сопоставления.

Таблица 9.16. *Примеры сопоставлений с помощью функции* `re.match()`

Аргументы	Сопоставление успешно?
`(r"abc", "abcdef", 0)`	Да
`(r"def", "abcdef", 0)`	Нет
`(r"ABC$", "abc\ndef", re.IGNORECASE)`	Нет
`(r"ABC$", "abc\ndef", re.I|re.MULTILINE)`	Да
`(r"abc$d", "abc\ndef", re.I|re.M)`	Нет
`(r"abc$d", "abc\ndef", re.I|re.M)`	Нет
`(r"\\**\\", "\\***\\", 0)`	Да

Таблица 9.16 *(окончание)*

Аргументы	Сопоставление успешно?
`("\\\\\\**\\\\", "\\****\\", 0)`	Да
`(r"", "\\****\\", 0)`	Да

Примечание

В новых версиях Python используется Unicode-версия модуля `re`. Документация по Python предупреждает, что в некоторых случаях при использовании этой версии возможно переполнение стека (это обещают исправить в будущих версиях). Старый модуль `re` (без поддержки Unicode) доступен под именем `pre`.

9.3.3. Модули *StringIO* и *cStringIO*

Модуль `StringIO` реализует ввод и вывод в строки, хранящиеся в памяти, но с использованием интерфейса файлового ввода/вывода. Используемый в модуле файлоподобный объект можно применять в большинстве операций над файлами.

Класс `StringIO` модуля `StringIO`. Перед началом работы с файлом его нужно открыть. Сделать это можно, вызвав на исполнение класс `StringIO`:

```
from StringIO import StringIO
fw = StringIO()     # объект для записи
fr = StringIO(s)    # объект для чтения из строки s
```

Работа с созданными таким образом файлоподобными объектами почти не отличается от работы с обычными файлами, за исключением следующих моментов:

- работа с реальным файлом может оказаться быстрее;
- не реализован метод `fileno()`;
- запись далеко за концом файла может вызвать порождение большого числа нулевых байтов в строке.

Повысить скорость можно за счет применения модуля `cStringIO`. Этот модуль написан на языке C и является упрощенным вариантом `StringIO`. Например, он не предоставляет класса `StringIO`, а только функцию для создания объектов.

В любой момент содержимое открытого файла, который на самом деле — строка в памяти, — можно извлечь с помощью метода `StringIO.getvalue()`:

```
from StringIO import StringIO
fw = StringIO()         # создать объект для записи
```

```
fw.seek(5)              # перейти на пятый байт файла
fw.write("a")           # записать "a" в файл
print `fw.getvalue()`   # прочитать и напечатать содержимое файла
```

Результат выполнения:

```
'\000\000\000\000\000a'
```

Из этого примера видно, что файл при необходимости растягивается, заполняясь нулевыми байтами, если позиция записи в файл устанавливается за концом файла.

По табл. 9.17 можно сравнить, какие атрибуты в каких файловых (и файлоподобных) объектах доступны.

Таблица 9.17. *Наличие атрибутов в различных файловых объектах*

Метод	Файл	cStringIO	StringIO
__class__	–	–	+
close()	+	+	+
closed	+	–	+
fileno()	+	–	–
flush()	+	+	+
getvalue()	–	+	+
isatty()	+	+	+
mode	+	–	–
name	+	–	–
read()	+	+	+
readinto()	+	–	–
readline()	+	+	+
readlines()	+	+	+
seek()	+	+	+
softspace	+	–	+
tell()	+	+	+
truncate()	+	+	+
write()	+	+	+
writelines()	+	+	+
xreadlines()	+	–	–

Модули `cStringIO` и `StringIO` применяются в тех случаях, когда к строке удобнее обращаться как к файлу. Такая потребность возникает обычно в случае, когда некоторый модуль может работать только с файлами, а нам нужно подать ему на вход строку.

9.3.4. Модуль *codecs* и работа с Unicode

До повсеместного применения Unicode еще далеко, поэтому, вероятно, необходимо объяснить, что же это такое. За тысячелетия развития, человеческая письменность создала тысячи различных символов: букв, цифр, иероглифов, научной нотации и т. д. Компьютеры же "мыслят" цифрами, представленными в виде битов, байтов (1 байт — 8 бит) и т. д. Для представления символов используются *кодировки*, т. е. однозначные соответствия между символом и его цифровым кодом. Тем самым, компьютер "знает", что если ему дан символ с кодом 65, то при выводе на экран дисплея (или принтер) ему нужно высветить (или напечатать):

```
000000000
000011000
000111100
001100110
001111110
001100110
011100111
000000000
```

До появления Unicode символы в компьютере кодировались одним байтом (а то и только семью битами). Один байт охватывает диапазон кодов от 0 до 255 включительно, и потому больше двух алфавитов (например, латинский и русский в обоих регистрах), цифр, знаков пунктуации и некоторого набора специальных символов охватить одной кодировкой не представлялось возможным. К бóльшему набору не были готовы и устройства вывода. Дело обострялось тем, что поначалу каждый производитель использовал свою кодировку для одного и того же алфавита. Например, до настоящего времени дожили целых пять кодировок букв кириллицы! Наверное, каждый пользователь русскоязычного Internet не раз видел в своем браузере или электронном письме пример несоответствия кодировок, когда разобрать смысл сообщения становится невозможно.

Появление Unicode вносит единую кодировку для символов всех языков мира. Это одновременно и большое облегчение, и некоторое неудобство. Облегчение состоит в том, что в одной Unicode-строке можно без всяких ухищрений написать рядом символы из совершенно разных языков. А неудобство связано с тем, что люди привыкли использовать однобайтовые кодировки, большинство текстовых редакторов и текстовых процессоров ори-

ентированы на них, во многих системах поддержка Unicode осуществляется лишь частично, т. к. требует огромной работы по разработке шрифтов. К счастью, символы в любой кодировке можно перевести в Unicode и обратно. Именно это и осуществляют кодеки, о которых пойдет речь ниже.

Модуль `codecs`, появившийся в Python 1.6, определяет базовые классы для стандартных *кодеков (кодировщиков-декодировщиков)* и дает доступ к внутреннему реестру кодеков, управляющему процессом их поиска. В модуле доступны следующие функции.

- `codecs.register(search_function)`

  Регистрирует функцию `search_function` для поиска кодеков. Такая функция должна принимать в качестве аргумента название кодировки строчными буквами и возвращать кортеж `(encoder, decoder, stream_reader, stream_writer)`. Функции `encoder` и `decoder` служат для кодирования и декодирования. Функции `stream_reader` и `stream_writer` служат для получения объектов, поддерживающих интерфейс классов `StreamReader` и `StreamWriter`. Они должны принимать в качестве аргументов поток ввода или вывода и необязательный аргумент `errors`, равный `"strict"`, `"ignore"` или `"replace"`, для указания способа обработки ошибок. Если `search_function` не находит кодека, она должна возвращать `None`.

- `codecs.lookup(encoding) -> (enc, dec, re, wr)`

  Возвращает кортеж функций, соответствующих кодировке `encoding`. Значения элементов кортежа объяснены выше. Пример в листинге 9.18 показывает изменение кодировки в зависимости от используемой платформы. Подразумевается, что при исполнении этого примера текст программы, исполняющейся под Windows, имеет кодировку cp1251, а при выполнении под UNIX используется koi8-r.

Листинг 9.18. Файл koiwin.py. Перекодировка

```
import codecs, sys
(koiencode,koidecode,sr,sw) = codecs.lookup('koi8-r')
(winencode,windecode,sr,sw) = codecs.lookup('cp1251')

if sys.platform.startswith("win"):
  print koiencode(windecode("Вопрос")[0])[0]
else:
  print winencode(koidecode("Вопрос")[0])[0]
```

- `codecs.open(filename, mode[, enc[, errors[, buffer]]]) -> file`

  Эта функция открывает файл для прозрачного кодирования/декодирования. Ее можно использовать для перекодирования файлов целиком,

как в листинге 9.19. Аргумент `errors` имеет то же значение, что и для `codecs.register()`, а `buffer` указывает размер буфера. По умолчанию применяется построчная буферизация (`buffer` равен единице).

Листинг 9.19. Файл recode.py. Универсальный перекодировщик

```
import codecs, sys
try:
  fromenc, toenc, fromfile, tofile = sys.argv[1:5]
except:
  print "Usage:\n  recode.py fromenc toenc fromfile tofile"
  sys.exit(0)

inp = codecs.open(fromfile, "rb", fromenc)
outp = codecs.open(tofile, "wb", toenc)
outp.write(inp.read())
```

❐ `codecs.EncodedFile(file, dataenc[, fileenc[, errors]]) -> file1`

Функция позволяет работать с файлом в кодировке fileenc, получая из него данные в кодировке dataenc.

В следующем примере мы записываем в файл текст в кодировке utf-8, тогда как в программе используется кодировка русских букв, принятая на данной платформе:

```
import sys, codecs
if sys.platform.startswith("win"):
  enc = "cp1251"
else:
  enc = "koi8-r"
# открываем файл и пишем в него:
file = codecs.EncodedFile(open("utf8.txt", "w"), enc, 'utf-8')
file.write("question - вопрос")
file.close()
# открываем файл и читаем из него:
file = codecs.EncodedFile(open("utf8.txt", "r"), enc, 'utf-8')
print file.read()
file.close()
```

Модуль `codecs` также определяет константы `BOM`, `BOM_BE`, `BOM_LE`, `BOM32_BE`, `BOM32_LE`, `BOM64_BE` и `BOM64_LE` для обозначения порядка байтов (BOM — byte order mark).

В табл. 9.18 перечислены другие классы и коротко описаны их назначения.

Таблица 9.18. *Классы, определенные в модуле* `codecs`

Класс	Описание
`Codec`	Определяет интерфейс кодеров/декодеров. Не хранит состояния. Имеет методы `encode()` и `decode()`. Эти методы принимают аргумент `input` со входным объектом, подлежащим (де)кодированию, и необязательный аргумент `errors` для указания способа обработки ошибок (см. табл. 8.4). Методы должны возвращать кортеж из выходного объекта и использованной при (де)кодировании длины (в некоторых кодировках последовательность данных не может оканчиваться на произвольном месте, а потому кодеки не гарантируют (де)кодирование от начала до конца заданного объекта)
`StreamWriter`	Подкласс класса `Codec`. В дополнение имеет методы `write()`, `writelines()` и `reset()`. Работает с файловыми объектами. Конструктор вызывается с аргументом `stream` (файлоподобный объект для вывода) и необязательным аргументом `errors`
`StreamReader`	Подкласс класса `Codec`. В дополнение имеет методы `read()`, `readline()`, `readlines()` и `reset()`. Конструктор вызывается с аргументом `stream` (файлоподобный объект для ввода) и необязательным аргументом `errors` для указания способа обработки ошибок
`StreamReaderWriter`	Определяет объединенный интерфейс классов `StreamReader` и `StreamWriter`. Конструктор вызывается с параметрами `stream`, `Reader`, `Writer` и `errors`, где `stream` — файловый объект ввода/вывода, `Reader` и `Writer` задают функции или классы, порождающие объекты с интерфейсами `StreamReader` и `StreamWriter` соответственно. Аргумент `errors` как и выше
`StreamRecoder`	Определяет интерфейс для обеспечения двустороннего кодирования/декодирования файлоподобного объекта, объединяя интерфейсы `StreamReader` и `StreamWriter`. Используется, например, в функции `codecs.EncodedFile()`

Примеры кодеков приведены в подкаталоге `encodings`, который находится в каталоге с файлами дистрибутива интерпретатора Python. В табл. 9.19 приведены названия кодировок кириллицы.

Таблица 9.19. *Кодировки для кириллицы*

Название	Комментарий
koi8-r	Самая ранняя из всех используемых кодировок кириллицы. Применяется, в основном, в системах UNIX и для кодирования электронной почты в России и СНГ, реже в Web-страницах. Обладает тем свойством, что при обнулении старшего бита текст можно разобрать даже на терминалах, не отображающих кириллицу. (Русские буквы заменяются их латинскими эквивалентами). Принята как стандарт Internet RFC-1489 по предложению А. Чернова. Кроме кириллицы, определяет также кодирование символов псевдографики
cp866	Так называемая альтернативная кодировка, использовавшаяся в DOS. Кроме кириллицы содержит символы псевдографики
cp1251	Кодировка Windows-1251, используемая в Windows. Буквы следуют по алфавиту, что упрощает сортировку строк в лексикографическом порядке
iso8859-5	Международная стандартная кодировка для кириллицы. Применяется в коммерческих базах данных и некоторых UNIX-системах, может использоваться для русификации Linux, поддерживается Windows. Буквы следуют в алфавитном порядке, относительное расположение всех символов такое же, как и в Unicode
mac_cyrillic	Кодировка кириллицы, принятая на Macintosh

Кроме того, сейчас все чаще применяется кодировка utf-8, в которой сочетаются возможности Unicode и совместимость с таблицей ASCII. Латинские буквы кодируются в utf-8 теми же кодами, что и в ASCII, а для кириллицы используются два байта. Символы некоторых других алфавитов могут потребовать и большего числа байтов.

Наконец, не стоит забывать, что Unicode-строки можно легко перекодировать в любую поддерживаемую кодировку методом `encode()`. Без этого Unicode-строка просто не выведется:

```
>>> s = u"\u041F\u041E\u0420\u0422"  # "ПОРТ"
>>> print s.encode("koi8-r")
ПОРТ
>>> print s

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> # ошибка преобразования в ASCII: код не в диапазоне от 0 до 128
```

9.3.5. Модуль *difflib*

Модуль `difflib` появился в Python 2.1 и используется для сравнения близких строк и формирования команд редактирования, приводящих одну строку к другой. (В системе UNIX имеется команда `diff`, которая позволяет сравнивать файлы с почти одинаковым содержимым и получать *патчи (patch, заплатка)*, т. е. набор инструкций для приведения одного файла к другому. Сокращение diff происходит от англ. слова *difference* — разница.)

Модуль `difflib` определяет следующую функцию

```
difflib.get_close_matches(word, variants[, n[, th]]) -> lst
```

которая выбирает из списка `variants` слова, наиболее "соответствующие" слову `word`. Возвращается не более `n` вариантов строк (по умолчанию `n` равно трем), а необязательный параметр `th` (число от `0` до `1` включительно) указывает допустимую степень близости: `0` — минимальная степень близости, `1` — точное соответствие (по умолчанию используется значение `0.6`).

```
>>> from difflib import *
>>> varnames = dir(__builtins__)
>>> print get_close_matches("max", varnames, 5, 0)
['max', 'map', 'complex', 'min', 'hex']
>>> print get_close_matches("max", varnames, 5, 0.5)
['max', 'map']
>>> print get_close_matches("max", varnames, 5, 1)
['max']
>>>
```

Вместо строк можно использовать другие последовательности.

В модуле также определен класс `SequenceMatcher` со следующим конструктором.

```
difflib.SequenceMatcher([isjunk[, s1[, s2]]]) -> smobj
```

где `isjunk` — функция, определяющая, является ли ее аргумент незначимым для сравнений или `None`, если все элементы являются значимыми; `s1`, `s2` — сравниваемые строки (по умолчанию пустые); `smobj` — объект класса `SequenceMatcher`.

В табл. 9.20 представлены методы объектов класса `SequenceMatcher`. Под `s1` и `s2` в описаниях понимаются заданные объекту последовательности.

Таблица 9.20. *Методы объектов класса* `SequenceMatcher`

Метод	Описание
`Set_seqs(s1, s2)`	Задает обе последовательности для сравнения
`Set_seq1(s1)`	Задает первую последовательность. При сравнении одной строки со многими, для смены последовательности лучше применять этот метод
`Set_seq2(s2)`	Задает вторую последовательность. При сравнении одной строки со многими, редко изменяемую последовательность лучше задавать этим методом, т. к. она, в отличие от первой, подвергается некоторой обработке, ускоряющей последующие сравнения
`find_longest_match (i1, i2, j1, j2) -> (i, j, k)`	Находит наиболее длинный блок, входящий как в `s1[i1:i2]`, так и в `s2[j1:j2]` (с учетом незначимых элементов, если таковые имеются). В кортеже-результате `i` — начало блока в `s1`, `j` — начало блока в `s2`, `k` — длина блока. Незначимые элементы не могут встречаться внутри блока, а только в его конце, и то в случае их идентичности. Если блоков одинаковой длины несколько, выбираются более ранние в `s1`. Если таких несколько в `s2`, выбирается более ранний
`get_matching_blocks () -> lst`	Возвращает список кортежей совпадающих блоков. Каждый элемент имеет формат `(i, j, k)`, как и результат `find_longest_match()`. Последний элемент особенный: это единственный элемент, в котором `k` равно нулю, а `i` и `j` равны длинам `s1` и `s2` соответственно
`get_opcodes() -> lst`	Возвращает список из пятиэлементных кортежей вида `(cmd, i1, i2, j1, j2)`, где `cmd` — символическое имя команды редактирования последовательности (табл. 9.21). Остальные элементы кортежа указывают индексы для срезов `s1[i1:i2]` и `s2[j1:j2]`, которые участвуют в команде. Если последовательно применить все указанные в списке команды к первой последовательности, должна получиться вторая последовательность
`ratio() -> r`	Подсчитывает индекс сходства двух строк, лежащий между нулем и единицей (включительно). Индекс равен `2.0*M/T`, где `T` — суммарная длина последовательностей, `M` — количество совпадающих блоков
`quick_ratio() -> r`	Вычисляет верхнюю границу `ratio()`
`real_quick_ratio() -> r`	То же, что и `quick_ratio()`, но работает быстрее

Таблица 9.21. *Условные обозначения команд редактирования*

Обозначение	Описание
"replace"	Заменить s1[i1:i2] на s2[j1:j2]
"delete"	Удалить s1[i1:i2]
"insert"	Вставить s2[j1:j2] на место s1[i1:i1]
"equal"	Ничего не делать, т. к. s1[i1:i2] равно s2[j1:j2]

В следующем примере демонстрируется работа метода get_opcodes(). Вместо строки в качестве s1 используется список символов. Это сделано для более удобной реализации команд. Кстати, все четыре команды свелись к смене среза в списке. Заметьте, что команды исполняются в обратной последовательности, чем та, что дает метод get_opcodes(). Это сделано для того, чтобы не пересчитывать индексы.

```
>>> import difflib, UserString
>>> s1 = map(None, "строка первая для примера")
>>> s2 = "еще строка вторая для примера"
>>> sm = difflib.SequenceMatcher(None, s1, s2)
>>> opc = sm.get_opcodes()
>>> opc.reverse()
>>> for (cmd, i1, i2, j1, j2) in opc:
...   print "%(cmd)s s1[%(i1)s:%(i2)s] s2[%(j1)s:%(j2)s]:" % vars(),
...   print '"%s" -> "%s"' % ("".join(s1[i1:i2]), s2[j1:j2])
...   s1[i1:i2] = map(None, s2[j1:j2])
...
equal s1[11:25] s2[15:29]: "ая для примера" -> "ая для примера"
delete s1[10:11] s2[15:15]: "в" -> ""
equal s1[9:10] s2[14:15]: "р" -> "р"
replace s1[7:9] s2[11:14]: "пе" -> "вто"
equal s1[0:7] s2[4:11]: "строка " -> "строка "
insert s1[0:0] s2[0:4]: "" -> "еще "
>>> print s2
еще строка вторая для примера
>>> print "".join(s1)
еще строка вторая для примера
```

Легко увидеть, что результат применения операций привел строку s1 к строке, равной s2.

9.3.6. Модуль *mmap*

Модуль `mmap` позволяет устраивать *отображение файлов в памяти (memory mapping)* с помощью объектов, которые имеют файловый интерфейс и понимают операции над изменяемыми строками фиксированной длины.

Для создания `mmap`-объектов используется функция `mmap.mmap()`.

- `mmap(fd, size[, flags, prot]) -> mmapobj`
- `mmap(fd, size[, tagname]) -> mmapobj`

Первый вариант используется на платформе UNIX, второй — в Windows. В качестве первого аргумента передается файловый дескриптор `fd` (его легко получить с помощью метода `fileno()` на реальном файле). Файл должен быть открыт для обновления (режим `"r+"`) . Параметр `size` задает количество байтов файла, которые следует отображать. Строкой `tagname` (под Windows) можно задать ярлык, под которым данный `mmap`-объект будет известен. Параметр `flags` (в UNIX) задает свойства отображения (с помощью `MAP`-констант), а `prot` — требуемую защиту памяти (`PROT`-константы) (табл. 9.22).

По умолчанию действуют `MAP_SHARED` и `PROT_READ|PROT_WRITE`.

Таблица 9.22. *Некоторые константы для задания свойств `mmap`-объектов*

Константа	Описание
`PROT_READ`	Страницы памяти могут быть прочитаны
`PROT_WRITE`	В страницы памяти можно писать
`PROT_EXEC`	Страницы памяти могут быть исполнены (не имеет смысла в Python)
`MAP_SHARED`	Все изменения отражаются на общей для процессов версии файла
`MAP_PRIVATE`	Все изменения отражаются только в версии файла для данного процесса (реальное создание индивидуальной версии происходит только при попытке записи)

Рассматриваемые здесь `mmap`-объекты позволяют использовать операции как над изменчивыми строками (взятие или присваивание элемента по индексу, выделение среза или присваивание срезу). Позволены только те операции, которые не изменяют длину объекта.

В табл. 9.23 описаны методы `mmap`-объектов.

Таблица 9.23. *Методы `mmap`-объектов*

Метод	Описание
`close()`	Закрывает файл

Таблица 9.23 *(окончание)*

Метод	Описание
`find(s[, start]) -> i`	Возвращает наименьший индекс, с которого начинается строка `s` или `-1`, если строка не встречается в файле. Параметр `start` указывает индекс, с которого следует начинать поиск, и по умолчанию равен `0`
`flush([offset, size])`	Сбрасывает изменения, сделанные в памяти, на диск. Если заданы `offset` и `size`, сбрасываются только байты в диапазоне `[offset:offset+size]`. По умолчанию сбрасывается вся отображаемая область файла
`move(dst, src, n)`	Копирует `n` байтов с позиции со смещением `src` в позицию со смещением `dst`
`read(n) -> s`	Читает до `n` байтов из файла
`read_byte() -> s`	Читает один байт из файла
`readline() -> s`	Читает строку из файла
`resize(newsize)`	Изменяет размер отображаемой на память области файла. Новый размер задается параметром `newsize`
`seek(pos [, whence])`	Устанавливает текущую позицию указателя в файле. Аналогичен соответствующему методу файловых объектов
`size()`	Возвращает длину исходного файла (эта длина может быть больше, чем размер отображаемой на память области)
`tell()`	Возвращает текущую позицию указателя. Аналогичен соответствующему методу файловых объектов
`write(s)`	Записывает байты строки `s` в файл
`write_byte(s)`	Записывает байт из односимвольной строки `s`

Следующий пример показывает работу с заголовком бинарного файла с использованием отображения на память:

```
>>> import mmap, sys, struct
>>> f = open("activate.wav", "r+")
>>> mm = mmap.mmap(f.fileno(), 48)
>>> print mm[36:40]
data
>>> print struct.unpack("l", mm[40:44])
(11682,)
>>> mm[40:44] = struct.pack("l", 12000)
>>> mm.flush()
0
```

9.4. Математические функции

В этом разделе представлены модули, основное содержание которых — математические функции для действительных и комплексных чисел, а также генераторы псевдослучайных чисел.

9.4.1. Модуль *math*

В модуле `math` собраны наиболее распространенные математические функции и константы, имеющиеся в стандартной библиотеке C и доступные из Python (табл. 9.24). Для комплексных чисел следует использовать модуль `cmath` *(см. разд. 9.4.2)*.

Таблица 9.24. *Функции и константы модуля* `math`

Функция или константа	Описание или эквивалентная формула
`acos(x)`	Арккосинус `x`
`asin(x)`	Арксинус `x`
`atan(x)`	Арктангенс `x`
`atan2(y,x)`	`atan(y/x)`
`ceil(x)`	Наименьшее целое, большее или равное `x`
`cos(x)`	Косинус `x`
`cosh(x)`	Гиперболический косинус `x`
`e`	Константа e
`exp(x)`	Экспонента (то есть `e**x`)
`fabs(x)`	Абсолютное значение `x`
`floor(x)`	Наибольшее целое, меньшее или равное `x`
`fmod(x,y)`	Остаток от деления `x` на `y`
`frexp(x)`	Возвращает мантиссу и порядок `x` как пару `(m, i)`, где `m` — число с плавающей точкой, а `i` — целое, такое, что `x = m * 2.**i`. Если `0`, возвращает `(0,0)`, иначе `0.5 <= abs(m) < 1.0`
`hypot(x,y)`	`sqrt(x*x + y*y)`
`ldexp(m,i)`	`m * (2**i)`
`log(x)`	Натуральный логарифм `x`
`log10(x)`	Десятичный логарифм `x`

Таблица 9.24 (окончание)

Функция или константа	Описание или эквивалентная формула
`modf(x)`	Возвращает пару `(y,z)` — целую и дробную часть `x`. Обе части имеют знак исходного числа
`pi`	Константа пи
`pow(x,y)`	`x**y`
`sin(x)`	Синус `x`
`sinh(x)`	Гиперболический синус `x`
`sqrt(x)`	Корень квадратный от `x`
`tan(x)`	Тангенс `x`
`tanh(x)`	Гиперболический тангенс `x`

Пример:

```
from math import *
print e, pi
d = 30              # градусов
print sin(d*pi/180), cos(d*pi/180), tan(d*pi/180)
proba = [-1.77, -1.5, -1.45, -0.5, -0.3, 0.5, 0.777, 1.1, 1.5]
print "proba:", proba
print "int:", map(int,proba)
print "ceil:", map(ceil,proba)
print "floor:", map(floor,proba)
print "round:", map(round,proba)
print "fabs:", map(fabs,proba)
print "fmod:", fmod(12.3,4)
print "frexp:", frexp(8)
print "modf:", modf(12.3)
print "log(exp):", log(exp(120))
```

Результат исполнения:

```
2.71828182846 3.14159265359
0.5 0.866025403784 0.57735026919
proba: [-1.77, -1.5, -1.45, -0.5, -0.29999999999999999, 0.5,
0.77700000000000002, 1.1000000000000001, 1.5]
int: [-1, -1, -1, 0, 0, 0, 0, 1, 1]
ceil: [-1.0, -1.0, -1.0, -0.0, -0.0, 1.0, 1.0, 2.0, 2.0]
floor: [-2.0, -2.0, -2.0, -1.0, -1.0, 0.0, 0.0, 1.0, 1.0]
```

```
round: [-2.0, -2.0, -1.0, -1.0, -0.0, 1.0, 1.0, 1.0, 2.0]
fabs: [1.77, 1.5, 1.45, 0.5, 0.29999999999999999, 0.5,
0.77700000000000002, 1.1000000000000001, 1.5]
fmod: 0.3
frexp: (0.5, 4)
modf: (0.30000000000000071, 12.0)
log(exp): 120.0
```

9.4.2. Модуль *cmath*

Модуль `cmath` аналогичен модулю `math` *(см. разд. 9.4.1)*, но его функции работают с комплексными числами (и возвращают комплексные числа, даже если результат можно выразить действительным числом). В табл. 9.25 содержатся описания объектов модуля `cmath`.

Таблица 9.25. *Функции и константы модуля* `cmath`

Функция или константа	Описание или эквивалентная формула
`acos(z)`	Арккосинус `z`
`asin(z)`	Арксинус `z`
`atan(z)`	Арктангенс `z`
`cos(z)`	Косинус `z`
`cosh(z)`	Гиперболический косинус `z`
`e`	Константа e (как действительное число)
`exp(z)`	Экспонента (`e**z`)
`log(z)`	Натуральный логарифм `z`
`log10(z)`	Десятичный логарифм `z`
`pi`	Константа пи (как действительное число)
`sin(z)`	Синус `z`
`sinh(z)`	Гиперболический синус `z`
`sqrt(z)`	Корень квадратный от `z`
`tan(z)`	Тангенс `z`
`tanh(z)`	Гиперболический тангенс `z`

Как обычно, рассмотрим пример:

```
from cmath import *
print sqrt(-1)
```

```
print sin(pi)
print acos(2)
print -exp(-1j*pi)
```

Результат выполнения:

```
1j
(1.22460635382e-16-0j)
1.31695789692j
(1+1.22460635382e-16j)
```

9.4.3. Модули *random* и *whrandom*

Модуль `random` предлагает генераторы *псевдослучайных чисел* для различных распределений случайных величин. Чаще всего используется равномерное распределение, для которого есть следующие функции.

- `choice(seq) -> item`

  Выбирает случайный элемент непустой последовательности (или возбуждает исключение `IndexError`).

- `random() -> rnd`

  Случайное число с плавающей точкой в диапазоне `[0.0 ... 1.0)`.

- `randrange([start,] stop[, step]) -> rnd`

  Случайное число из полуоткрытого диапазона `start—stop` с шагом `step`. Равносильно `choice(range(start, stop, step))`.

- `uniform(a, b) -> rnd`

  Псевдослучайное число из диапазона `[a...b)`. Равносильно числу `random()*(b-a)+a`.

- `seed([num])`

  Задать новую затравку. Если число не задано, затравка получит значение на основе текущего времени.

- `shuffle(seq)`

  Перемешивает элементы (изменяемой) последовательности (на месте). Возвращает `None`.

Также доступны генераторы для некоторых других распределений.

- `betavariate(alpha, beta) -> rnd`

  Бета-распределение, у которого `alpha > -1`, `beta > -1`. Возвращаемые значения между `0` и `1`.

- `cunifvariate(mean, arc) -> rnd`

  Круговое равномерное распределение. Здесь `mean` обозначает средний угол, `arc` — амплитуду (оба значения в радианах), от `0` до пи. Возвращаемые значения лежат между `mean-arc/2` и `mean+arc/2`.

- `expovariate(лямбда) -> rnd`

  Экспоненциальное распределение. Параметр `лямбда` равен обратному значению требуемого среднего. Возвращаемые значения — от `0` до положительной бесконечности.

- `gamma(alpha, beta) -> rnd`

  Гамма-распределение с `alpha > -1`, `beta > 0`.

  `gauss(mu, sigma) -> rnd`

  `normvariate(mu, sigma) -> rnd`

  Гауссово (нормальное) распределение. Здесь `mu` — среднее, `sigma` — среднеквадратическое отклонение (`sigma > 0`). Первая из указанных функций работает несколько быстрее.

- `lognormvariate(mu, sigma) -> rnd`

  Логнормальное распределение. Натуральный логарифм случайной величины с данным распределением имеет нормальное (гауссово) распределение со средним `mu` и среднеквадратическим отклонением `sigma`.

- `paretovariate(alpha) -> rnd`

  Распределение Парето с параметром `alpha`.

- `weibullvariate(alpha, beta) -> rnd`

  Распределение Вейбулла.

- `vonmisesvariate(mu, kappa) -> rnd`

  Распределение Мизеса.

Стандартный для Python интерфейс к классу генератора псевдослучайных чисел следующий (в качестве названия класса здесь используем `generator` — как в модуле `random`). В модуле `whrandom`, реализующем алгоритм Вичмана-Хилла, генератор называется `whrandom`. Изучение исходного кода модуля `random` показывает, что он базируется на `whrandom`.

```
random.generator([seed]) -> generator_object
```

Конструктор класса. Здесь `seed` — затравка (для конкретных видов генераторов могут быть свои правила в отношении затравок).

Методы `choice()`, `uniform()`, `randrange()`, `random()`, `seed()` аналогичны описанным выше одноименным функциям.

Использование различных объектов-генераторов приводит к независимым последовательностям псевдослучайных чисел.

9.5. Время и календарь

Ниже представлены два стандартных модуля Python для работы со временем и календарем.

В сети Internet можно найти модуль расширения `mxDateTime` (от eGenix.com Software GmbH), которого, например, требуют модули, связывающие Python с базами данных. Этот модуль представляет собой гораздо бóльший набор функций для работы со временем и датами, чем модули, описываемые в этом разделе.

9.5.1. Модуль *time*

Модуль `time` позволяет работать с датами и временем и фактически является интерфейсом к соответствующим системным функциям языка C.

Вслед за C и UNIX, время в Python считается посекундно, начиная с определенной точки, называемой эпохой. В UNIX эпоха начинается с начала 1970 года, когда время от начала эпохи равно нулю. Эпоха не бесконечна и может определяться длиной машинного слова. На 32-разрядных архитектурах эпоха заканчивается обычно в 2038 году (когда для хранения количества секунд не хватит 31 двоичного разряда) или в 2068 году (не хватит и 32 разрядов). Функции модуля обычно требуют указания полного года. Двузначные годы переводятся в четырехзначные по следующей схеме, удовлетворяющей стандарту POSIX: 00—68 становятся 2000—2068, а 69—99 — 1969—1999. Значения 100—1899 не принимаются.

Время в системе UNIX считается по Гринвичу (GMT, Greenwich Mean Time), или как его теперь называют — *универсальное координируемое время (UTC, Universal Time Coordinated)*. Это связано с тем, что в этой многопользовательской системе различные пользователи могут находиться в различных временных зонах и возникла бы неразбериха, например, со временем создания файлов, записями в логи и т. п. Кроме того, в различных странах некоторое время в году действуют различные поправки к местному времени, известные как *летнее время (DST, Daylight Saving Time)*. Правила перехода на летнее время и обратно могут меняться. К счастью, библиотека языка достаточно корректно хранит эти правила и проводит необходимые вычисления для конвертирования местного времени из UTC и обратно.

Время в секундах от начала эпохи не самый удобный для человека способ представления времени. В Python для представления времени в развернутом виде используется кортеж из девяти элементов (для краткости назовем его кортежем времени). Элементы кортежа описаны в табл. 9.26.

Таблица 9.26. *Строение кортежа времени*

Индекс	Поле	Диапазон значений
0	Год	0—99, 1900—2068 и далее
1	Месяц	1—12
2	День	1—31
3	Час	0—23
4	Минута	0—59
5	Секунда	0—61
6	День недели	0—6, начиная с понедельника
7	День года	1—366
8	Флаг летнего времени	0, 1 или –1 (если нет информации)

В модуле определены следующие переменные и константы.

- `time.accept2dyear`

  Логическое значение. Если истина, функции модуля будут обрабатывать обозначение года из двух цифр, иначе — нет. По умолчанию истина. Переменная окружения `PYTHONY2K`, имеющая непустое значение, устанавливает `accept2dyear` в `0`. Значение можно менять из программы.

- `time.altzone`

  Смещение местного летнего времени в секундах от времени в UTC. Восточнее Гринвича отрицательно. Например, для Москвы равно `-14400`.

- `time.daylight`

  Логическое значение. Равно `0`, если летнее время не определено, и `1` — в противном случае. Для временной зоны MSK/MSD равно `1`.

- `time.timezone`

  Смещение местного времени в секундах от времени в UTC. Восточнее Гринвича отрицательно. Например, для Москвы равно `-10800`.

- `time.tzname`

  Название временных зон для зимнего и летнего местного времени. Для Москвы: `('MSK', 'MSD')`.

В модуле определены следующие функции.

- `time.asctime(time_tuple) -> string`

  `time.ctime (secs) -> string`

  Преобразуют кортеж времени `time_tuple` или число секунд с начала эпохи `secs` в строку из 24 символов вида `'Mon Mar 5 19:42:48 2001'`.

- `time.clock () -> float`

  Возвращает время центрального процессора в секундах, потраченное на исполнение данной программы к настоящему моменту. С помощью этой функции можно засекать время выполнения программ и их отдельных частей.

- `time.gmtime (secs) -> time_tuple`

  Преобразует время в секундах с начала эпохи в кортеж времени UTC.

- `time.localtime (secs) -> time_tuple`

  Преобразует время в секундах с начала эпохи в кортеж времени для местного времени. Флаг летнего времени устанавливается при необходимости.

- `time.mktime (time_tuple) -> secs`

  Преобразует местное время, записанное в виде кортежа времени, в секунды от начала эпохи (число с плавающей точкой). Если флаг летнего времени не определен, можно использовать значение `-1` (хотя есть риск, что для одного часа в году, при переходе с летнего времени на зимнее, результат будет некорректен). При неправильных параметрах возбуждает исключение `OverflowError`.

- `sleep (secs)`

  Приостанавливает исполнение программы на заданное (необязательно целое) число секунд. Указанный период времени не является гарантированным, т. к., с одной стороны, приостановленное приложение может получить сигнал, для обработки которого "сон" будет прерван. С другой стороны, система может выполнять какие-либо другие приложения и программа может не получить управление строго через заданное время. Вызов `sleep()` предпочтительнее традиционного для "однозадачного" программирования цикла задержки, т. к. высвобождает процессорное время для других задач.

- `strftime (fmt, time_tuple) -> string`

  Преобразует кортеж времени `time_tuple` в строку `string` согласно заданному формату `fmt`. Представление некоторых элементов зависит от текущей культурной среды. Обозначения, которые можно использовать в строке формата, приведены в табл. 9.27.

Таблица 9.27. *Обозначения строки формата*

Символ	Значение	Примечание
`%a`	Сокращение дня недели	
`%A`	Полное название дня недели	
`%b`	Сокращение названия месяца	

Таблица 9.27 *(окончание)*

Символ	Значение	Примечание
`%B`	Полное название месяца	
`%c`	Представление даты и времени	
`%d`	День месяца, число от 1 до 31	
`%H`	Час от 0 до 23	
`%I`	Час от 1 до 12	
`%j`	День года от 1 до 366	
`%m`	Номер месяца от 1 до 12	
`%M`	Минута от 0 до 59	
`%p`	Указание AM/PM	Для текущей культурной среды
`%S`	Секунда от 0 до 61	Не стоит забывать, что иногда добавляют 1–2 секунды для коррекции календаря
`%U`	Номер недели в году от 0 до 53	Неделя считается начинающейся с воскресенья. Дни до первого воскресенья принадлежат неделе 0
`%w`	День недели от 0 (воскресенье) до 6	
`%W`	Номер недели в году от 0 до 53	Неделя считается начинающейся с понедельника. Дни до первого понедельника принадлежат неделе 0
`%x`	Представление даты	
`%X`	Представление времени	
`%y`	Год без указания века от 0 до 99	
`%Y`	Год (четырехзначный)	
`%Z`	Временная зона	
`%%`	Знак процента	

❒ `strptime (string[, fmt]) -> time_tuple`

Разбирает строку `string` с записанной датой и временем согласно формату `fmt`, возвращая кортеж времени. Формат тот же, что и для `strftime`. По умолчанию используется формат `"%a %b %d %H:%M:%S %Y"`. В случае неудачи возбуждается `ValueError`. Функция полностью зависит от соответствующей реализации одноименной функции из библиотеки C для данной платформы, а на некоторых платформах даже отсутствует

(Windows). Некоторые из реализаций содержат ошибки. Используйте с осторожностью.

- `time () -> secs`

Возвращает число секунд от начала эпохи в UTC.

В табл. 9.28 обобщены функции по преобразованию времени.

Таблица 9.28. *Функции для преобразования между различными представлениями времени*

Преобразование	Из `secs`	Из `time_tuple`	Из `string`
В `secs`	—	`mktime()`	—
В `time_tuple`	`gmtime()`, `localtime()`	—	`strptime()`
В `string`	`ctime()`	`asctime()`, `strftime()`	—

В следующем примере присутствуют некоторые из этих преобразований. Обратите внимание на пример восстановления последних трех элементов кортежа времени:

```
>>> from time import *
>>> today = localtime(time())
>>> print "Сегодня:", strftime("%d.%m.%Y %H:%M:%S", today)
Сегодня: 07.07.2001 12:03:24
>>> date1G = 1000000000
>>> print "Круглая дата:", localtime(date1G)
Круглая дата: (2001, 9, 9, 5, 46, 40, 6, 252, 1)
>>> guess = (2001, 7, 26, 10, 20, 35, -1, -1, -1)
>>> print "Не все данные известны:", guess
Не все данные известны: (2001, 7, 26, 10, 20, 35, -1, -1, -1)
>>> guess = localtime(mktime(guess))
>>> print "Но их можно восстановить:", guess
Но их можно восстановить: (2001, 7, 26, 10, 20, 35, 3, 207, 1)
```

Нужно иметь в виду, что восстановление флага летнего времени не всегда однозначно. Пример в листинге 9.20 покавывает ситуацию, происходящую при переходе на зимнее время. Целых два часа в году невозможно определить точное время исходя только из местного времени. Поэтому, если вы хотите хранить или передавать время некоторого события, лучше всего либо использовать время в секундах от фиксированного момента в прошлом (тре-

тья колонка), либо каждый раз указывать признак летнего времени (последний элемент в кортеже времени).

Листинг 9.20. Петля времени из-за перехода на зимнее время

```
from time import *
loopdate = (2001, 10, 28, 1, 20, 0, 6, 301, 1)
output_fmt = "%-19s %-35s %-15s"
print output_fmt % ("дата и время", "кортеж времени", "секунды")
for dt in range(0, 4*3600, 3600/2):
  unixtime = mktime(loopdate) + dt
  ltime = localtime(unixtime)
  verbose = strftime("%d.%m.%Y %H:%M:%S", ltime)
  print output_fmt % (verbose, ltime, unixtime)
```

Результат выполнения:

```
дата и время        кортеж времени                          секунды
28.10.2001 01:20:00 (2001, 10, 28, 1, 20, 0, 6, 301, 1) 1004217600.0
28.10.2001 01:50:00 (2001, 10, 28, 1, 50, 0, 6, 301, 1) 1004219400.0
28.10.2001 02:20:00 (2001, 10, 28, 2, 20, 0, 6, 301, 1) 1004221200.0
28.10.2001 02:50:00 (2001, 10, 28, 2, 50, 0, 6, 301, 1) 1004223000.0
28.10.2001 02:20:00 (2001, 10, 28, 2, 20, 0, 6, 301, 0) 1004224800.0
28.10.2001 02:50:00 (2001, 10, 28, 2, 50, 0, 6, 301, 0) 1004226600.0
28.10.2001 03:20:00 (2001, 10, 28, 3, 20, 0, 6, 301, 0) 1004228400.0
28.10.2001 03:50:00 (2001, 10, 28, 3, 50, 0, 6, 301, 0) 1004230200.0
```

В следующем примере используется русская культурная среда для вывода элементов даты.

```
import time, string, locale
locale.setlocale(locale.LC_CTYPE, "ru_RU.KOI8-R")
locale.setlocale(locale.LC_TIME, "ru_RU.KOI8-R")
now = time.localtime(time.time())
month = string.lower(time.strftime("%B", now))
print "Сегодня %s %s." %(now[1], month)
```

Результат выполнения:

```
Сегодня 7 июля.
```

Еще одна особенность, которую следует иметь в виду авторам программ реального времени, заключается в том, что системное время (которое использует

функция time.time) в некоторых системах может совершать легкие прыжки вперед или назад, которые вызваны необходимостью поддерживать это время точным. Таким образом, time() <= time() не всегда выполняется. Кроме того, не стоит забывать, что системное время можно корректировать вручную или с использованием сервера времени, что может еще более серьезно повлиять на работу программ, для которых критично неубывание времени.

9.5.2. Модуль *calendar*

В модуле calendar содержатся полезные функции для формирования и печати календаря. Здесь мы не будем рассматривать функций calendar.prcal, calendar.prmonth и calendar.prweek — их применение ограничено выводом на стандартный вывод и представляет интерес только для работающих в Python в интерактивном режиме.

Так в модуле нумеруются дни недели (табл. 9.29).

Таблица 9.29. *Дни недели*

Имя	Значение	День недели
calendar.MONDAY	0	Понедельник
calendar.TUESDAY	1	Вторник
calendar.WEDNESDAY	2	Среда
calendar.THURSDAY	3	Четверг
calendar.FRIDAY	4	Пятница
calendar.SATURDAY	5	Суббота
calendar.SUNDAY	6	Воскресенье

Для работы с календарем будут полезны следующие функции.

- calendar.setfirstweekday(weekday)

  Устанавливает какой день недели считать первым. По умолчанию 0 (понедельник). Для некоторых стран будет удобнее 6 (воскресенье).

- calendar.firstweekday() -> int

  Возвращает номер дня недели, считающийся первым.

- calendar.isleap(year) -> 0, 1

  Логическое значение. Является ли год year високосным?

- calendar.leapdays(year1, year2) -> int

  Подсчитывает количество високосных лет с year1 по year2, не включая year2.

- `calendar.weekday(year, month, day) -> int`

  Возвращает день недели для любого года, начиная с 1970.

- `calendar.monthrange(year, month) -> (wd, days)`

  Возвращает кортеж, где `wd` — день недели первого числа указанного месяца, `days` — количество дней в месяце.

- `calendar.monthcalendar(year, month) -> matrix`

  Возвращает матрицу `matrix` (список списков недель) со всеми числами месяца.

Следующий пример использует некоторые из описанных выше функций:

```
>>> import pprint
>>> from calendar import *
>>> setfirstweekday(MONDAY)  # неделю начинаем с понедельника
>>> print firstweekday()  # первый день недели - понедельник
0
>>> print map(isleap, range(1998,2002))  # високосные годы
[0, 0, 1, 0]
>>> print leapdays(1998, 2002)  # число доп. дней
1
>>> print weekday(1974, 2, 20)  # какой день недели был тогда?
2
>>> print monthrange(2001, 2)  # день недели первого дня и кол-во дней
(3, 28)
>>> pprint.pprint(monthcalendar(2001,2))  # календарик
[[0, 0, 0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9, 10, 11],
 [12, 13, 14, 15, 16, 17, 18],
 [19, 20, 21, 22, 23, 24, 25],
 [26, 27, 28, 0, 0, 0, 0]]
```

9.6. Массивы и структуры

Два модуля, описанные в этом разделе, позволяют работать со следующими типами данных: массивами (наборами однородных элементов) и записями (наборами разнородных элементов фиксированного формата). Эти модули могут потребоваться, например, при необходимости чтения или записи файлов, имеющих специальный бинарный формат.

9.6.1. Модуль *array*

Язык Python не имеет встроенного типа данных для *массивов*. Списки являются более гибким типом данных, но работа с ними требует большего числа вычислений. Массивы менее гибки (все элементы одного типа и размера), но зато легко моделируются плоской моделью памяти и потому выборка элемента массива — гораздо более быстрая операция, чем аналогичная операция над элементом списка. Модуль `array` является интерфейсом к (одномерным) массивам языка C с элементами различных типов. Конечно, для более сложных приложений лучше воспользоваться массивами из пакета Numeric Python *(см. гл. 11)*.

Конструктор массива вызывается следующим образом.

```
array.array(typecode [, initializer]) -> array
```

где `typecode` — код типа элементов (табл. 9.30); `initializer` — последовательность для инициализации массива.

Таблица 9.30. *Типы элементов массива*

Код типа	Тип C	Мин. размер в байтах
`'c'`	`Char`	1
`'b'`	`signed int`	1
`'B'`	`unsigned int`	1
`'h'`	`signed int`	2
`'H'`	`unsigned int`	2
`'i'`	`signed int`	2
`'I'`	`unsigned int`	2
`'l'`	`signed int`	4
`'L'`	`unsigned int`	4
`'f'`	`float`	4
`'d'`	`double`	8

Функция `array.array()` возвращает объект `array`, методы которого описаны ниже.

- Методы, которые аналогичны соответствующим методам для списков:

  `extend()`, `append()`, `count()`, `index()`, `reverse()`, `pop()`, `remove()`.

- Методы для работы с отдельными элементами и срезами (выделение среза порождает новый массив).

- Методы, приведенные в табл. 9.31, позволяют вводить и выводить значения из массива в строки, списки, файлы.

Таблица 9.31. *Методы объекта-массива*

Название метода	Назначение атрибута
`fromstring(string)`	Добавляет элементы в конец массива из строки
`fromlist(list)`	Добавляет элементы в конец массива из списка
`fromfile(file)` или `read(file)`	Добавляет элементы в конец массива из открытого файла
`tostring()`	Выводит массив в строку
`tolist()`	Выводит массив в список
`tofile()` или `write()`	Выводит массив в файл

Здесь следует заметить, что значения из массива и в массив в этих шести методах передаются в их машинном представлении.

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> b = a[1:3]
>>> a[1:2] = array.array('i', [6])
>>> for i in a: print i,
...
1 6 3 4 5
>>> print `a.tostring()`
'\001\000\000\000\006\000\000\000\003\000\000\000\004\000\000\000\005\000
\000\000'
>>> print b
array('i', [2, 3])
```

9.6.2. Модуль *struct*

С помощью модуля `struct` можно преобразовывать значения переменных языка Python в структуры на языке C и обратно. Для представления структур C в виде последовательности байтов используются строки. Битовые поля в структурах не поддерживаются. Информация о структуре задается строкой формата.

В строке формата могут употребляться следующие символы:

- Самый первый символ (он может и отсутствовать) определяет порядок байтов в машинном слове (на компьютерах разных архитектур этот порядок может быть различным) (табл. 9.32).

- Остальные символы указывают трактовку конвертируемых аргументов, (табл. 9.33).

Таблица 9.32. *Модификатор порядка байтов*

Символ	Порядок байтов	Выравнивание
`@`	Родной	Родное
`=`	Родной	Стандартное
`<`	Сначала младшие байты	Стандартное
`>`	Сначала старшие байты	Стандартное
`!`	Принятый в сети (сначала старшие байты)	Стандартное

Таблица 9.33. *Типы элементов структуры*

Символ	Тип C	Тип Python	Примечание
`x`	Незначащий байт	Не требует аргумента	
`c`	`char` (символ)	Односимвольная строка	
`b`	`signed byte` (байт со знаком)	Целое число	
`B`	`unsigned byte` (байт без знака)	Целое число	
`h`	`short` (короткое целое)	Целое число	
`H`	`unsigned short` (короткое целое без знака)	Целое число	
`i`	`int` (целое)	Целое число	
`I`	`unsigned int` (целое без знака)	Целое или длинное целое	
`l`	`long` (длинное целое)	Целое	
`L`	`unsigned long` (длинное целое без знака)	Длинное целое	
`f`	`float` (число с плавающей точкой)	Число с плавающей точкой	
`d`	`double` (то же, с двойной точностью)	Число с плавающей точкой	
`s`	`string` (массив символов)	Строка	Числовой префикс задает длину

Таблица 9.33 *(окончание)*

Символ	Тип C	Тип Python	Примечание
p	pascal string (паскалевская строка)	Строка	Числовой префикс задает длину
P	Указатель	Целое, достаточно длинное для хранения указателя	Порядок байтов всегда родной
пробел			Игнорируется

Перед каждым символом можно добавить числовой префикс, который говорит о числе аргументов соответствующего типа. Число аргументов и их типы должны строго соответствовать строке формата, в противном случае возбуждается исключение `struct.error`. Строки дополняются незначащими символами или усекаются до требуемой длины.

В модуле доступны следующие функции:

- `struct.calcsize(fmt) -> int`
- `struct.pack(fmt, v1, v2, ...) -> string`
- `struct.unpack(fmt, string) -> (v1, v2, ...)`

где `fmt` — строка, содержащая формат структуры; `v1, ...` — члены структуры, подлежащие конвертированию или результат конвертирования; `string` — строка, представляющая структуру побайтно.

Функция `struct.calcsize()` вычисляет размер структуры по заданному формату. `struct.pack()` записывает ("упаковывает") структуру в строку string, а `struct.unpack()` читает ("распаковывает") структуру из строки, согласно строке формата `fmt`. Функции `struct.pack()` и `struct.unpack()` применяются, в основном, для чтения и записи бинарных файлов.

```
import struct
def pr_hex(comment, s):
    """ красиво печатаем последовательность байтов """
    print comment,
    for c in s:
        print hex(ord(c)),
    print

print "calcsize 12p: ", struct.calcsize("12p")
print "Размеры различных типов данных:"
for m in "xcbBhHiIlLfd":
    print "%s: %s, "%(m, struct.calcsize(m)),
    if m in ["H","d"]: print
```

```
print "Результаты упаковки:"
pr_hex( " 2l:", struct.pack(" 2l", 1, -1) )
pr_hex( ">2l:", struct.pack(">2l", 1, -1) )
pr_hex( "<2l:", struct.pack("<2l", 1, -1) )
pas_str = struct.pack("4p", "123456")
pr_hex( "pas_str:", pas_str)
print "pas_str:", struct.unpack("4p", pas_str)
```

Результат выполнения:

```
calcsize 12p:  12
Размеры различных типов данных:
x: 1,  c: 1,  b: 1,  B: 1,  h: 2,  H: 2,
i: 4,  I: 4,  l: 4,  L: 4,  f: 4,  d: 8,
Результаты упаковки:
 2l: 0x1 0x0 0x0 0x0 0xff 0xff 0xff 0xff
>2l: 0x0 0x0 0x0 0x1 0xff 0xff 0xff 0xff
<2l: 0x1 0x0 0x0 0x0 0xff 0xff 0xff 0xff
pas_str: 0x3 0x31 0x32 0x33
pas_str: ('123',)
```

Примечание

Данный пример отработал на архитектуре Intel i386, для компилятора Gnu C. На других платформах результат может оказаться иным.

Следующий пример показывает ввод/вывод для бинарных файлов.

```
import struct
fmt = "bb4s"
# Пишем в файл
f = open("somefile.tmp", "wb")  # "b" указано для совместимости
f.write(struct.pack(fmt, 1, 2, "abcd"))
f.close()
# Читаем из файла
f = open("somefile.tmp", "rb")
a, b, s = struct.unpack(fmt, f.read(struct.calcsize(fmt)))
f.close()
print a, b, s
```

Результат выполнения:

```
1 2 abcd
```

9.7. Взаимодействие с операционными системами (файлы, процессы)

При реализации практически любого проекта возникает необходимость взаимодействия с операционной системой, в которой выполняется интерпретатор Python. Не все операционные системы предоставляют одинаковые сервисы, поэтому не все функции, описываемые ниже, будут работать на всех компьютерных платформах. Тем не менее, приведенные модули более или менее не зависят от используемой платформы.

9.7.1. Модуль *os*

Модуль `os` служит для работы с сервисами операционной системы. Этот модуль скрывает, насколько это возможно, зависимость от конкретной ОС. Для задач, требующих сервисов операционной системы, желательно использовать функции этого модуля, а не платформно-зависимых: `'posix'`, `'nt'`, `'dos'`, `'mac'`, `'os2'`, `'ce'`, `'java'`,...

Узнать, какая у вас платформа, можно подав следующие команды:

```
>>> import os
>>> print os.name
posix
```

Этот пример был исполнен в системе Linux, поэтому `os.name` равен `"posix"`, что означает, что в качестве платформы использована система, работающая по стандарту POSIX.

Модуль `os` достаточно обширный и рассмотрение его функций будет удобно разбить на несколько пунктов.

Информация о системе

В табл. 9.34 приведены символьные константы, относящиеся к элементам записи путей к файлам и указаны значения для платформ POSIX, Windows, DOS, MacOS.

Таблица 9.34. *Константы, определяющие элементы пути к файлу, на разных платформах*

Константа	Что обозначает	POSIX	Windows	Macintosh
`os.curdir`	Текущий каталог	`"."`	`"."`	`":"`
`os.pardir`	Родительский каталог	`".."`	`".."`	`"::"`
`os.sep`	Разделитель элементов пути	`"/"`	`"\"`	`":"`

Таблица 9.34 *(окончание)*

Константа	Что обозначает	POSIX	Windows	Macintosh
os.altsep	Альтернативный разделитель элементов пути		"/"	
os.pathsep	Разделитель путей в списке путей	":"	";"	
os.defpath	Список путей по умолчанию	':/bin:/usr/bin'		
os.linesep	Признак окончания строки	"\n"	"\n\r"	"\r"

Параметры процесса

Программа на Python запускается в операционной системе как отдельный процесс. С помощью описанных ниже функций можно считывать и устанавливать различные значения, относящиеся к процессу и к среде, в которой он исполняется.

Очень важную роль играет словарь по имени `sys.environ`: он содержит значения переменных окружения, которые используются интерпретатором Python и программой на Python для различных целей. Например, через переменные окружения CGI-сценарий получает информацию от Web-сервера. Чтение по ключу из словаря `sys.environ` соответствует функции `getenv()` библиотеки C, а запись значения в словарь (где она поддерживается) — функции `putenv()`. Следующий пример распечатывает значения всех переменных окружения и может использоваться, например, в качестве тестового CGI-сценария:

Листинг 9.21. Простейший отладочный CGI-сценарий

```
#!/usr/bin/env python
import os
print "Content-Type: text/plain\n\n"
for k, v in os.environ:
  print k + "=" + `v`
```

В табл. 9.35 приведены различные функции, работающие с характеристиками процесса. За исключением функций `getpid()`, `putenv()`, `strerror()` и `umask()` (которые доступны и на платформе Windows), функции, перечисленные в этой таблице, работают только на UNIX-системах. Все упоминания о процессе относятся к текущему процессу, откуда функции вызываются.

Таблица 9.35. *Функции для работы с характеристиками процесса*

Функция	Описание
`os.ctermid()`	Возвращает имя файла, соответствующее терминалу, на котором запущен процесс
`os.getegid()`, `os.geteuid`	Возвращает действующие номер группы и пользователя в системе
`os.getgid()`, `os.getuid`	Возвращает номер группы и пользователя в системе
`os.getgroups()`	Возвращает номера групп, от имени которых процесс может действовать
`os.getlogin()`	Получает входное имя, под которым работает данный процесс (обычно имя пользователя, запустившего процесс)
`os.getpid()`	Возвращает номер процесса
`os.getppid()`	Возвращает номер родительского процесса
`os.putenv(name, value)`	Присваивает переменной окружения `name` значение `value`
`os.setegid(egid)`, `os.seteuid(euid)`	Устанавливает действующие номер группы и пользователя (используется для отказа от лишних привилегий)
`os.setgid(gid)`, `os.setuid(uid)`	Устанавливает номер группы и пользователя в заданные значения
`os.strerror(code)`	Возвращает текст ошибки с номером `code`
`os.umask(new) -> old`	Назначает новое значение маски (для назначения прав на доступ к файлу) и возвращает старое значение

Здесь мы не упомянули (достаточно редкие) функции `getpgrp()`, `setpgrp()`, `setreuid()`, `setregid()` и `setsid()`. О них лучше справиться в документации по UNIX.

Следующий пример показывает, как влияет `umask` на права доступа к файлам, которые создает Python-программа:

```
import os
def mk_and_chk_file(filename):
  f = open(filename, "w")
  f.write(".")
  f.close()
  os.system("ls -l " + filename)

os.umask(000)
mk_and_chk_file("f1")
```

```
os.umask(002)
mk_and_chk_file("f2")
os.umask(077)
mk_and_chk_file("f3")
```

Результат выполнения:

```
-rw-rw-rw-    1 rnd      rnd             1 Jul  4 00:33 f1
-rw-rw-r--    1 rnd      rnd             1 Jul  4 00:33 f2
-rw-------    1 rnd      rnd             1 Jul  4 00:33 f3
```

Маску лучше задавать в восьмеричной системе счисления, т. к. каждый бит ее означает запрет на определенную операцию (`r` — чтение, `w` — запись, `x` — исполнение) определенной категорией пользователей: младшие три бита соответствуют всем пользователям (очень часто 7 или 6 — им обычно незачем не только записывать в ваши файлы, но и читать их!), следующие три бита определяют возможности доступа для членов групп, к которым вы принадлежите, а старшие три бита определяют права доступа для владельца файла (обычно — 0). Таким образом, файл `f1` будет доступен всем для чтения, записи и исполнения, `f2` не будет доступен чужим пользователям для записи, а `f3` сможет читать, записывать и исполнить только владелец.

Следующий пример показывает, что не все операции доступны обычным пользователям:

```
>>> try:
...   os.setuid(0)  # пытаемся стать супер-пользователем
... except os.error, errmsg:
...   print errmsg
...
[Errno 1] Operation not permitted
>>> # операция не разрешена
```

Создание и управление файловым объектом

В этом разделе описаны операции, в которых участвуют так называемые *файловые дескрипторы* — целые числа, под которыми операционная система учитывает открытые данным процессом файлы.

Предупреждение

Операции с файловыми дескрипторами (ФД) являются более низкоуровневыми по сравнению с операциями с файловыми объектами, и потому их стоит применять только в случаях, когда это действительно необходимо.

В табл. 9.36 перечислено большинство операций модуля `os`, работающих с ФД, и указана их доступность на основных платформах.

Таблица 9.36. *Функции модуля* `os` *для работы с файловыми дескрипторами (ФД)*

Функция	Назначение	Доступна
`os.close(fd)`	Закрывает файл с данным дескриптором	MUW
`os.dup(fd) -> fd1`	Возвращает дубликат ФД	MUW
`os.dup2(fd, fd2)`	Делает дубликат ФД `fd` в `fd2`, закрыв предварительно `fd2`	UW
`os.fstat(fd) -> st`	Возвращает статус ФД (по аналогии с `stat()`)	UW
`os.fstatvfs(fd) -> st`	Возвращает статус файловой системы, содержащей файл с данным дескриптором (по аналогии с `statvfs()`)	U
`os.ftruncate (fd, length)`	Обрубает файл, связанный с ФД `fd`, до длины `length`	U
`os.isatty(fd) -> 0, 1`	Является ли файл с ФД `fd` tty-устройством	U
`os.lseek(fd, pos, how)`	Аналог метода `seek()` для файловых объектов	MUW
`os.open(file, flags[, mode]) -> fd`	Открывает файл с именем `file`, применив флаги `flags` и правами доступа `mode`. Флаги описаны в табл. 9.37. По умолчанию права доступа к файлу mode имеют значение `0777` с занулением битов, присутствующих в маске `umask`	MUW
`os.openpty() -> (master_fd, slave_fd)`	Открывает пару дескрипторов, образующих псевдотерминальную пару	U
`os.pipe() -> (r, w)`	Возвращает пару ФД для чтения (`r`) и записи (`w`) из созданного канала	UW
`read(fd, n) -> s`	Читает данные из файла, связанного с ФД `fd`, не более `n` байтов за раз	MUW
`os.ttyname(fd) -> name`	Возвращает имя связанного с `fd` терминала (или возбуждает исключение `OSError`, если `fd` не связан с tty-устройством)	U
`os.write(fd, str)`	Записывает `str` в файл с ФД `fd` и возвращает количество реально записанных в файл байтов	MUW

Примечание

В таблице используются обозначения: U — UNIX, W — Windows, M — Macintosh.

В табл. 9.37 представлены константы модуля os, которые могут использоваться функцией os.open() в качестве флагов. Как обычно, для задания некоторого набора флагов используется побитовое ИЛИ. Не все флаги или их комбинации действуют на всех системах.

Таблица 9.37. *Флаги, применяющиеся в функции* os.open()

Флаг	Назначение
os.O_RDONLY	Открытие файла только для чтения
os.O_WRONLY	То же, только для записи
os.O_RDWR	То же, для записи и чтения
os.O_APPEND	Файл открывается в режиме добавления. Перед каждой записью указатель устанавливается на конец файла
os.O_CREAT	При попытке открытия несуществующего файла, файл будет создан
os.O_TRUNC	При открытии существующего файла, файл будет обрезан до нулевой длины
os.O_EXCL	При совместном применении с O_CREAT в случае, когда файл уже существует, возбуждается исключение OSError. Тем самым гарантируется, что существующий файл не будет испорчен при создании нового файла
os.O_NONBLOCK, os.O_NDELAY	Открытие файла без блокировки (и, соответственно, без ожиданий, когда файл освободится)
os.O_RSYNC, os.O_SYNC, os.O_DSYNC	Различные варианты синхронного ввода/вывода. Вызывающий изменения в файле процесс будет останавливаться, пока заданные изменения (в статусе и/или содержимом файла) не будут записаны на физический носитель
os.O_NOCTTY	Если файл — tty-устройство, то оно не станет контролирующим терминалом процесса, даже в случае отсутствия у процесса такового

В следующем примере мы получаем файловый дескриптор для нового файла.

```
import os
fd = os.open("file.dat", os.O_WRONLY|os.O_TRUNC|os.O_CREAT|os.O_SYNC)
for i in xrange(3000):
  os.write(fd, str(i))
os.close(fd)
```

В данном случае файл открывается в синхронном режиме, и потому каждый вызов os.write() ждет, пока данные будут реально записаны на диск. Поэтому приведенная программа будет выполняться крайне медленно по срав-

нению с вариантом без os.O_SYNC. Тем не менее, в некоторых приложениях необходим именно синхронный режим записи.

Работа с файлами и каталогами

Если до этого мы рассматривали файлы с точки зрения их содержимого, то приведенные ниже функции позволяют работать с файлом как с целым: проверять существование, копировать, удалять, изменять статус. В табл. 9.38 перечислены все входящие в эту категорию функции. (Разумеется, пути path задаются в виде строк.)

Таблица 9.38. *Функции для работы с файлами*

Функция	Описание	Платформы
os.access(path, okeys)	Проверка доступности файла или каталога, заданного в строке path. Режим запрашиваемого доступа задается значением okeys, составленных комбинацией (как обычно, побитовым ИЛИ) флагов os.F_OK (файл существует), os.R_OK (из файла можно читать), os.W_OK (в файл можно писать) и os.X_OK (файл можно исполнять, каталог можно просматривать)	UW
os.chdir(path)	Делает каталог path текущим рабочим каталогом	MUW
os.getcwd()	Получает текущий рабочий каталог	MUW
os.chmod(path, mode)	Устанавливает режим доступа к path в значение mode. Последнее можно составить заданием комбинации флагов, описанных в табл. 9.49 *(см. разд. 9.7.11)*. Будьте внимательны: chmod() не добавляет режим к существующему, а устанавливает его целиком	UW
os.chown(path, uid, gid)	Изменяет идентификаторы владельца (uid) и группы (gid) файла или каталога path	UW
os.link(file, lnk)	Делает lnk жесткой ссылкой на file. При этом file должен существовать, а lnk существовать не должно	U
os.listdir(dir)	Возвращает список файлов в каталоге dir. В список не входят специальные значения "." и ".."	MUW
os.lstat(path)	Аналог os.stat(), не следующий по символическим ссылкам (то есть дающий информацию по самому файлу-ссылке, а не по файлу, на который указывает ссылка)	U

Таблица 9.38 (продолжение)

Функция	Описание	Платформы
`os.mkfifo(path [, mode])`	Создает именованный канал `path`. По умолчанию применяется `mode`, равный `0666` (это дает всем права на чтение и запись). Пример использования именованного канала дан ниже	U
`os.mkdir(path [, mode])`	Создает каталог `path`. По умолчанию `mode` равен `0777`, т. е. `S_IRWXU|S_IRWXG|S_IRWXO` *(см. разд. 9.7.11)*	MUW
`os.makedirs (path[, mode])`	Аналогично `os.mkdir()`, но создает все необходимые каталоги, если они не существуют. Возбуждает исключение, если последний каталог уже существует	MUW
`os.readlink (path)`	Возвращает строку с путем, на который указывает символическая ссылка `path`	U
`os.remove(path)`, `os.unlink(path)`	Удаляет файл `path`. Для удаления каталогов используются `os.rmdir` и `os.removedirs`	
`os.rmdir(path)`	Удаляет каталог `path`, если он пуст	MUW
`os.removedirs (path)`	Как и `os.rmdir()`, но удаляет `path` до первого непустого каталога. В случае, если самый правый каталог в указанном пути непустой — возбуждает исключение `OSError`	MUW
`os.rename(src, dst)`	Переименовывает файл или каталог `src` в `dst`	
`os.renames(src, dst)`	Аналог `os.rename()`, который создает все необходимые каталоги для пути `dst` и удаляет пустые каталоги пути `src`	MUW
`os.stat(path)`	Возвращает информацию о пути `path` в виде не менее чем десятиэлементного кортежа. *(см. разд. 9.7.11)*	MUW
`os.statvfs( path)`	Возвращает информацию о файловой системе, в которой находится `path`, в виде десятиэлементного кортежа. Расшифровка значений дана в модуле `statvfs`, который в данной книге не рассматривается	U
`os.symlink (file, lnk)`	Создает символическую ссылку `lnk`, указывающую на `file`	U
`os.tempnam ([dir[, prefix]])`	Формирует уникальное имя для временного файла в заданном каталоге `dir` и, возможно, с префиксом `prefix`. Не забывайте удалять временные файлы после использования. *См. также разд. 9.7.12*, где дано более удобное решение для создания временных файлов	UW

Таблица 9.38 *(окончание)*

Функция	Описание	Платформы
`os.tmpnam()`	То же, но использует принятый в системе каталог для временных файлов. Константа `os.TMP_MAX` дает максимально возможное число временных файлов, на которое способна функция `os.tmpnam`	UW
`os.utime (path, times)`	Устанавливает значения времен последней модификации (`mtime`) и доступа к файлу (`atime`). Если `times` равен `None`, в качестве времен берется текущее время. В других случаях `times` рассматривается как двухэлементный кортеж `(atime, mtime)`. (Для получения `atime` и `mtime` некоторого файла можно использовать `os.stat()`)	MUW

Примеры, приведенные в листингах 9.22—9.24, показывают достаточно типичные применения перечисленных функций:

Листинг 9.22. Существует ли файл?

```
import os
if os.access("file.dat", os.F_OK):
  print "существует"
else:
  print "не существует"
```

В следующих двух листингах создается именованный канал между двумя процессами, через который из одного процесса в другой передаются некоторые данные. Эти примеры работают только в системах UNIX или Linux. Поместите файлы europe.py и america.py в один каталог и запустите их на исполнение с разных консолей (сначала лучше запустить europe.py).

Листинг 9.23. Файл europe.py. Передающая сторона именованного канала

```
import os, time, sys
os.mkfifo("transatlantic")    # создаем канал
channel = open("transatlantic", "w", 1)  # открываем его для записи
for t in range(5):
  line = time.ctime(time.time()) + "\n"
  channel.write(line)  # пишем в канал текущее время
  sys.stderr.write(line)
```

```
  time.sleep(1)  # процесс засыпает на секунду
channel.close()
```

Листинг 9.24. Файл america.py. Принимающая сторона именованного канала

```
import os, sys
# ждем создания канала
while not os.access("transatlantic", os.R_OK):
  pass
channel = open("transatlantic", "r", 1)  # открываем канал
# читаем построчно и выводим на stderr
while 1:
  line = channel.readline()
  if not line:
    break
  sys.stderr.write(line)
channel.close()
# удаляем канал
os.unlink("transatlantic")
```

Обратите внимание, что функция `open()` открывает канал с использованием построчной буферизации (ей задан третий аргумент — `1`). Это сделано для придания наглядности процессу передачи строк из одного объекта в другой.

Следующий листинг содержит модуль `dirtree` с функцией `dir_tree()` для вывода каталога в виде дерева:

Листинг 9.25. Файл dirtree.py. Модуль `dirtree`

```
import os

def dir_tree(dir, path=None, prefix="", last=1):
  """dir_tree(dir)
  Вывод каталога dir в виде дерева
  """
  path = path or dir
  if os.path.exists(path):
    isdir = os.path.isdir(path)
    print prefix + ("|", "`")[last] + "-" + dir + ("", "/")[isdir]
    if isdir:
      files = os.listdir(path)
```

```
      if files:
        prefix = prefix + ("| ", "  ")[last]
        files.sort()
        for file in files[:-1]:
          dir_tree(file, path + "/" + file, prefix, 0)
        dir_tree(files[-1], path + "/" + files[-1], prefix, 1)

def main():
  """Модуль в качестве главной программы выводит в виде
  дерева каталоги, заданные в командной строке"""
  import sys
  for dir in sys.argv[1:]:
    dir_tree(dir)

if __name__ == '__main__':
  main()
```

В функции `dir_tree` производится обход каталогов и их вывод с одновременным рисованием древесной структуры псевдографикой. Аргументы функции означают следующее: `dir` — имя каталога, `path` — путь к каталогу или файлу (накапливается по мере погружения в каталоги), `prefix` — префикс из псевдографики для прорисовки дерева, `last` — флаг, указывающий, что файл последний и выводится немного по-другому. В частности, `("|", "`")[last]` как раз и выбирает, что вывести перед именем в зависимости от значения флага, `0` или `1`.

Пример в листинге 9.26 использует только что приведенный модуль `dirtree` для иллюстрации манипуляций с каталогами:

Листинг 9.26. Манипулирование каталогами

```
import dirtree, os

commands = [
  """os.mkdir("aaa")""",
  """os.makedirs("aaa/bbb/ccc/ddd")""",
  """os.makedirs("aaa/bbb/zzz")""",
  """os.renames("aaa/bbb/zzz", "aaa/zzz")""",
  """os.removedirs("aaa/bbb/ccc/ddd")""",
  """os.removedirs("aaa/zzz")""",
]
```

```
for cmd in commands:
  exec cmd
  print cmd, ":"
  dirtree.dir_tree("aaa")
```

Результат выполнения:

```
os.mkdir("aaa") :
`-aaa/
os.makedirs("aaa/bbb/ccc/ddd") :
`-aaa/
  `-bbb/
    `-ccc/
      `-ddd/
os.makedirs("aaa/bbb/zzz") :
`-aaa/
  `-bbb/
    |-ccc/
    | `-ddd/
    `-zzz/
os.renames("aaa/bbb/zzz", "aaa/zzz") :
`-aaa/
  |-bbb/
  | `-ccc/
  |   `-ddd/
  `-zzz/
os.removedirs("aaa/bbb/ccc/ddd") :
`-aaa/
  `-zzz/
os.removedirs("aaa/zzz") :
```

Следующий пример посвящен изменению времени создания файла. Такая задача возникает, например, в архиваторах. Архиватор должен придать файлу его оригинальную дату последней модификации (`mtime`). В примере ниже мы устанавливаем дату последнего доступа к содержимому файла и дату последней модификации на неделю назад от текущей даты:

Листинг 9.27. Установка даты изменения и последнего доступа к файлу

```
import os, stat
from time import *
```

```
def show_times(file):
  print file, ":"
  st = os.stat(file)
  print "Последний доступ: ", ctime(st[stat.ST_ATIME])
  print "Модифицирован:    ", ctime(st[stat.ST_MTIME])
  print "Изменение статуса:", ctime(st[stat.ST_CTIME])

file = "file.dat"
show_times(file)
os.utime(file, (time()-3600*24*7, time()-3600*24*7))
show_times(file)
```

Результат выполнения:

```
file.dat :
Последний доступ:  Fri Jul  6 23:38:53 2001
Модифицирован:     Fri Jul  6 23:38:53 2001
Изменение статуса: Fri Jul  6 23:38:53 2001
file.dat :
Последний доступ:  Fri Jun 29 23:39:00 2001
Модифицирован:     Fri Jun 29 23:39:00 2001
Изменение статуса: Fri Jul  6 23:39:00 2001
```

Создание и управление процессами

В основу многих современных операционных систем заложена *многозадачность (multitasking)*, т. е. возможность параллельного исполнения задач. Даже если процессор один, операционная система может достаточно быстро переключать задачи, чтобы создалось впечатление их параллельной работы.

Здесь мы рассмотрим один из вариантов управления параллельным исполнением программ: управление *процессами. (Об альтернативном подходе — потоках управления см. разд. 9.7.13.)*

Каждый процесс в системе имеет свое адресное пространство и не может мешать другим процессам. Операционная система разделяет процессор (или процессоры) компьютера между исполняющимися процессами в соответствии с их приоритетом. Общение между процессами возможно благодаря сокетам, потокам ввода/вывода, именованным потокам и другим средствам ввода/вывода. *(Пример общения процессов был приведен в статье "Работа с файлами и каталогами" в разд. 9.7.1.)*

Интерпретатор Python, исполняющий программу, тоже является процессом, о чем говорилось в *статье "Параметры процесса" разд. 9.7.1.* Здесь же мы

рассмотрим, как управлять другими процессами. В табл. 9.39 содержатся функции, относящиеся к созданию и управлению процессами.

Таблица 9.39. *Функции для создания и управления процессами*

Функция	Назначение	Платформы
`os.abort()`	Вызывает для текущего процесса сигнал `SIGABRT` *(см. разд. 9.7.10)*. В UNIX процесс завершается с записью файла core, в Windows процесс возвращает код завершения 3	UW
`os._exit(n)`	Завершает процесс с кодом завершения `n`. При этом не вызываются процедуры сброса буферов, закрытия файлов и т. п. В отличие от функции `exit()`, используется для выхода из дочерних процессов (то есть порожденных с помощью `fork()`)	U
`exec*`	Это семейство функций для замещения текущего процесса другим описано отдельно	UW
`os.fork()`	Разветвляет процесс на родительский и дочерний (см. пример в листинге 9.28). При этом дочерний процесс отличается от родительского только некоторыми деталями (`pid`, `ppid` и другими, в основном, системными "паспортными" данными). Возвращает `0`, если в дочернем процессе, и `pid` дочернего процесса, если в родительском процессе	U
`os.forkpty()`	То же, но создает для дочернего процесса новый псевдотерминал (`pty`). Возвращает (`0`, ?) если в дочернем процессе, и (`pid`, `fd`) дочернего процесса, если в родительском процессе, где `fd` — файловый дескриптор управляющей части псевдотерминала	U
`os.kill(pid, signal)`	Посылает процессу `pid` сигнал `signal`. *(См. разд. 9.7.10.)*	U
`os.nice(incr)`	Изменяет приоритет текущего процесса на значение `incr`. Возвращает результирующий приоритет. Приоритет лежит в диапазоне от –20 до 19 включительно. Чем больше это значение, тем меньше приоритет	U
`os.plock(op)`	Закрепляет сегменты программы в оперативной памяти. Значение `op` можно узнать в файле sys/lock.h	U
`os.startfile (path)`	Запускает на исполнение файл, обозначенный в пути `path`	W

Таблица 9.39 *(окончание)*

Функция	Назначение	Платформы
`os.system(cmd)`	Выполняет командную строку `cmd` в отдельной оболочке (`shell`). Аналогично вызову `system()` библиотеки языка C. Возвращаемое значение зависит от используемой платформы. Изменения в `os.environ`, `sys.stdin` и т. п. не отражаются на программе, исполняемой в оболочке. При использовании вывода команды `system()` в CGI-сценариях могут возникнуть проблемы из-за буферизации `sys.stdout` (см. листинг 9.30)	
`spawn*`	Это семейство функций для порождения процессов описано отдельно	UW
`os.times()`	Возвращает кортеж из пяти элементов, содержащий время в секундах: время работы процесса, время работы ОС (по обслуживанию процесса), время работы дочерних процессов, время работы ОС для дочерних процессов, время в секундах от фиксированного момента в прошлом (например, старт системы). Уточнить детали можно в руководствах по используемой ОС	UW
`os.wait() -> (pid, status)`	Ждет завершения дочернего процесса. Возвращает `pid` этого процесса и индикатор завершения `exitstatus`. Младший байт, т. е. `exitstatus & 0xFF`, содержит номер сигнала, из-за которого процесс был завершен, а старший — `exitstatus >> 8` — статус завершения (если младший байт был нулем). Старший бит младшего байта указывает на то, что дамп памяти процесса был записан в файл core	U
`os.waitpid(pid, options) -> (pid, status)`	Ждет завершения дочернего процесса `pid`. Возвращаемый результат такой же, что и у `wait()`. В `options` можно использовать `os.WNOHANG`, чтобы текущий процесс не зависал, если статус дочернего процесса сейчас недоступен.	

В табл. 9.40 показаны различные функции семейств `exec*` и `spawn*` и их аргументы.

Функции `exec*` замещают текущий процесс на другой, указанный в `path`, с аргументами командной строки `args` (или `arg0, arg1, ...`) и переменными окружения, заданными словарем `env`.

Аналогично, функции spawn* порождают новый процесс. Аргумент mode управляет режимом создания процесса и может принимать значения: P_WAIT, P_NOWAIT, P_NOWAITO, P_OVERLAY, P_DETACH. (Последние два значения только для Windows.) Таким образом, текущий процесс может ждать (wait) или не ждать (nowait) окончания работы порождаемого процесса.

Таблица 9.40. *Аргументы функций* exec* *и* spawn*

Функции	Аргументы
os.execl, os.execlp	(path, arg0, arg1, ...)
os.execle, os.execlpe	(path, arg0, arg1, ..., env)
os.execv, os.execvp	(path, args)
os.execve, os.execvpe	(path, args, env)
os.spawnl, os.spawnlp	(mode, path, arg0, arg1, ...)
os.spawnle, os.spawnlpe	(mode, path, arg0, arg1, ..., env)
os.spawnv, os.spawnvp	(mode, path, args)
os.spawnve, os.spawnvpe	(mode, path, args, env)

Примечание

Как мы видим, l и v отличают способ задания аргументов (в строке вызова либо в списке). Указание p производит над аргументами процедуру развертывания, как если бы это происходило в командной строке. Например, "*" заменяется на список файлов в текущем каталоге. Буква e говорит о том, что последний аргумент дает словарь, задающий переменные окружения нового процесса.

Пример в листинге 9.28 иллюстрирует параллельную асинхронную работу двух процессов, возникающих в результате применения функции fork().

Листинг 9.28. Файл fork_example.py. Пример применения os.fork()

```
#!/usr/bin/python
import os, random, time, sys
pid = os.fork()  # "раздвоение личности"
if pid:
  # мы в родительском процессе
  letter, nice, delta = "P", 0, 0
else:
  # дочернем процессе
  letter, nice, delta = "Д", 10, 0.5
```

```
# остальное и там, и тут
os.nice(nice)  # меняем приоритет процесса
for i in range(10):
  sys.stdout.write(letter)  # пишем свою букву
  sys.stdout.flush()  # сбрасываем буфер
  time.sleep(delta+random.random())  # спим случайное время
```

Результат выполнения:

```
РДРДРРДРРРДРРДРДДДДД
```

Здесь сто́ит заметить, что результат выполнения недетерминирован, т. к. получен в результате вывода двух асинхронных процессов в один поток.

Замечание

Очень рекомендуем найти возможность и повозиться с функцией `fork()` и параллельными процессами. Конечно, не все программы можно безболезненно разделять на несколько процессов указанным способом, т. к. не всегда ресурс, разделяемый между процессами, можно разделять (открытые файлы, окна графического интерфейса, сетевые соединения...), т. к. может получиться такая же мешанина, как в примере выше.

В следующем простом примере вызывается команда `dir`:

Листинг 9.29. Пример применения функции `os.system()`

```
import os
os.system("dir")
```

В листинге 9.30 приведен пример CGI-сценария, в котором используются системные команды, вызываемые функцией `os.system()`. В указанном виде CGI-сценарий может не сработать на некоторых UNIX-системах из-за того, что вызываемые команды могут начать писать в стандартный вывод раньше операторов `print` из-за буферизации. Что, разумеется, тут же вызывает ошибку 500 на Web-сервере.

Листинг 9.30. Проблема с буферизацией вывода в CGI-сценарии

```
#!/usr/bin/python
import os
print "Content-Type: text/html\n\n"
print "<HTML><HEAD><TITLE>WRONG CGI</TITLE><BODY><PRE>"
os.system("date")
os.system("ls")
print "</PRE></BODY></HTML>"
```

Решение, представленное в листинге 9.31, состоит в сбрасывании буфера потока стандартного вывода.

Листинг 9.31. Решение проблемы в CGI-сценарии

```
#!/usr/bin/python
import os, sys
print "Content-Type: text/html\n"
print "<HTML><HEAD><TITLE>RIGHT CGI</TITLE><BODY><PRE>"
sys.stdout.flush()
os.system("/bin/date")
os.system("ls")
sys.stdout.flush()
print "</PRE></BODY></HTML>"
```

9.7.2. Модуль *os.path*

Функции для работы с путями к файлам и каталогам выделены в отдельный модуль — `os.path`. Заметьте, что часть функций работает с путями чисто "теоретически", тогда как в других функциях указанный в аргументе путь должен обязательно существовать, иначе возбуждается исключение `os.error`.

- `os.path.abspath(path) -> path1`

  Возвращает нормализованный абсолютный путь `path1` для `path`.

- `os.path.basename(path) -> s`

  Возвращает базовое имя файла или каталога, т. е. без ведущего к ним пути. Равносильна `os.path.split(path)[1]`.

- `os.path.commonprefix(list_of_paths) -> prefix`

  Пользуясь списком путей, дает общий префикс всех этих путей. Результат необязательно является путем. Если список пуст, возвращается пустая строка.

- `os.path.dirname(path) -> dirpath`

  Возвращает имя каталога, в котором находится заданный путем `path` файл или каталог. Равносильна `os.path.split(path)[0]`.

- `os.path.exists(path) -> 0, 1`

  Возвращает истину, если путь существует.

- `os.path.expanduser(path) -> path1`

  Подставляет домашний каталог данного пользователя вместо начальной тильды "~", а сочетание `~user` — на домашний каталог указанного пользователя `user`. Путь возвращается неизменным, если замены не удались или путь не начинается на тильду "~".

- `os.path.expandvars(path) -> path1`

  Подставляет в путь значения заданных переменных окружения. То есть вместо `$name` и `${name}` подставляется значение переменной окружения `name`. Неправильно заданные имена и ошибки в написании оставляют путь неизменным.

- `os.path.getatime(path) -> atime`

  Возвращает время последнего доступа к файлу или каталогу `path`. Время выражается в секундах от начала эпохи *(см. разд. 9.5.1)*.

- `os.path.getmtime(path) -> mtime`

  Возвращает время последней модификации файла или каталога. Время выражается в секундах от начала эпохи.

- `os.path.getsize(path) -> bytes`

  Возвращает размер файла или каталога в байтах.

- `os.path.isabs(path) -> 0, 1`

  Возвращает истинное значение, если путь является абсолютным.

- `os.path.isfile(path) -> 0, 1`

  Проверяет, является ли указанный путь файлом.

- `os.path.isdir(path) -> 0, 1`

  Проверяет, является ли указанный путь путем к каталогу.

- `os.path.islink(path) -> 0, 1`

  Проверяет, является ли указанный путь символической ссылкой.

- `os.path.ismount(path) -> 0, 1`

  Проверяет, является ли указанный путь точкой монтирования другой файловой системы.

- `os.path.join(path1[, path2[, ...]]) -> joinedpath`

  Склеивает кусочки пути вместе, вставляя необходимые разделители. Если какой-нибудь путь начинается от корня, все предыдущие кусочки пути отбрасываются.

- `os.path.normcase(path) -> path1`

  На системах, не чувствительных к смене регистра букв, приводит путь к строчным буквам. На Windows также преобразует "/" в "\\".

- `os.path.normpath(path) -> path1`

  Нормализует путь, убирая лишние `".."` и `"."`. На Windows также преобразует `"/"` в `"\\"`. Регистр букв не изменяется.

- `os.path.samefile(path1, path2) -> 0, 1`

  Проверяет, является ли path1 и path2 одним и тем же файлом. Работает на Macintosh и UNIX.

- `os.path.sameopenfile(f1, f2) -> 0, 1`

  Проверяет, являются ли `f1` и `f2` одними и теми же файловыми объектами. Работает на Macintosh и UNIX.

- `os.path.samestat(stat1, stat2) -> 0, 1`

  Возвращает истинное значение, если кортежи `stat2` и `stat1` (например, результаты функции `os.stat()`) относятся к статусной информации одного и того же файла.

- `os.path.split(path) -> (dirname, basename)`

  Разбивает путь на два куска: `dirname` (имя каталога) и `basename` (базовое имя файла или каталога). Часть `basename` не может содержать разделителей каталогов (вроде `"/"`), но может быть пустой, если `path` пуст или оканчивается на разделитель каталогов. Часть `dirname` пуста только если `path` пуст или не содержит разделителей каталогов.

- `os.path.splitdrive(path) -> (drive, rest)`

  На системах, где диск указывается отдельным префиксом, разделяет указание диска и остальной участок пути.

- `os.path.splitext(path) -> (head, ext)`

  Разбивает путь на основную часть и расширение файла.

- `os.path.walk(path, visit, arg)`

  Служит для рекурсивного обхода каталога `path`. В каждом посещаемом каталоге вызывается функция `visit` с аргументами `(arg, dirname, names)`, где `arg` — значение аргумента `arg` из вызова `os.path.walk()`; `dirname` — имя каталога, для которого вызывается функция `visit`; `names` — имена файлов и каталогов в этом каталоге, подлежащих дальнейшему обходу. Список, связанный с именем `names`, можно изменять, изменяя или удаляя отдельные элементы или срезы.

Следующий пример показывает, что общий префикс, выделяемый функцией `os.path.commonprefix()`, может не являться путем к реальному имени файла:

```
>>> import os
>>> print os.path.commonprefix(["/abck/def", "/abcd/ef", "/abcdef"])
/abc
```

9.7.3. Модуль *dircache*

Модуль `dircache` позволяет получать список файлов в каталоге и использует кэширование этих данных для более быстрого повторного доступа. Модуль предоставляет следующие функции.

- `dircache.listdir(path) -> list`

  Возвращает список файлов в указанном каталоге.

- `dircache.reset()`

  Сбрасывает содержимое кэша.

- `dircache.annotate(dirname, list)`

  Добавляет `"/"` к именам каталогов в списке `list`.

9.7.4. Модуль *errno*

Модуль `errno` описывает константы для значений стандартных (POSIX) номеров ошибок, возвращаемых системными функциями, а также словарь для текстовых сообщений об этих ошибках. В табл. 9.41 представлены некоторые константы модуля `errno`, представляющие номера стандартных системных ошибок.

Таблица 9.41. *Стандартные системные ошибки*

Имя	Сообщение	Перевод
`E2BIG`	Arg list too long	Список аргументов слишком велик
`EACCES`	Permission denied	Доступ запрещен
`EAGAIN`	Try again	Попытайтесь еще
`EBADF`	Bad file number	Неподходящий файловый дескриптор
`EBADMSG`	Not a data message	Плохое сообщение
`EBUSY`	Device or resource busy	Устройство или ресурс заняты
`ECHILD`	No child processes	Нет дочерних процессов
`EDEADLK`	Resource deadlock would occur	Избежание тупиковой ситуации
`EDOM`	Math argument out of domain of func	Аргумент функции не из области возможных значений
`EEXIST`	File exists	Файл существует
`EFAULT`	Bad address	Плохой адрес
`EFBIG`	File too large	Файл слишком велик
`EINPROGRESS`	Operation now in progress	Операция уже идет

Таблица 9.41 *(продолжение)*

Имя	Сообщение	Перевод
EINTR	Interrupted system call	Прерванный вызов функции
EINVAL	Invalid argument	Неверный аргумент
EIO	I/O error	Ошибка ввода/вывода
EISDIR	Is a directory	Является каталогом
EMFILE	Too many open files	Слишком много открытых файлов
EMLINK	Too many links	Слишком много ссылок
EMSGSIZE	Message too long	Слишком длинное сообщение
ENAMETOOLONG	File name too long	Слишком длинное имя файла
ENFILE	File table overflow	В системе слишком много открытых файлов
ENODATA	No data available	Нет данных
ENODEV	No such device	Нет такого устройства
ENOENT	No such file or directory	Нет такого файла или каталога
ENOEXEC	Exec format error	Ошибка формата исполняемого файла
ENOLCK	No locks available	Нет доступных блокировок
ENOMEM	Out of memory	Нет памяти
ENOSPC	No space left on device	Нет свободного места на устройстве
ENOSYS	Function not implemented	Функция не реализована
ENOTDIR	Not a directory	Некаталог
ENOTEMPTY	Directory not empty	Каталог непуст
ENOTTY	Not a tty	Неверная команда управления вводом/выводом (не tty-устройство)
ENXIO	No such device or address	Нет такого устройства или адреса
EOVERFLOW	Value too large for defined data type	Слишком большое значение для данного типа данных
EPERM	Operation not permitted	Операция не разрешена
EPIPE	Broken pipe	Разрыв канала
EPROTO	Protocol error	Ошибка протокола
ERANGE	Math result not representable	Ошибка представления результата
EROFS	Read-only file system	Файловая система только для чтения

Таблица 9.41 *(окончание)*

Имя	Сообщение	Перевод
ESPIPE	Illegal seek	Некорректная установка позиции в файле
ESRCH	No such process	Нет такого процесса
ETIMEDOUT	Connection timed out	Соединение прервано по тайм-ауту
ETXTBSY	Text file busy	Текстовый файл занят
EUSERS	Too many users	Слишком много пользователей
EXDEV	Cross-device link	Ссылка за границу устройства

Словарь `errno.errorcode` позволяет переводить номера ошибок в символические имена ошибок:

```
>>> import os, errno
>>> try:
...   os.rmdir("~/tmp")
... except OSError, errobj:
...   print errobj
...   print errno.errorcode[errobj.errno]
...
[Errno 2] No such file or directory: '~/tmp'
ENOENT
```

Текстовое сообщение по номеру ошибки можно узнать с помощью функции `os.strerror`:

```
>>> import os, errno
>>> print os.strerror(errno.ENOENT)
No such file or directory
```

9.7.5. Модуль *getopt*

Программы, запускаемые из командной строки, зачастую позволяют указывать те или иные ключи (опции), влияющие на логику работы программы. Модуль `getopt` позволяет анализировать аргументы командной строки и выделять заданные в ней опции.

Модуль предоставляет функцию `getopt.getopt()`.

```
getopt.getopt(rawargs, options[, long_options]) -> (optlist, args)
```

где `rawargs` — аргументы командной строки (можно задать `sys.argv[1:]`); `options` — описание однобуквенных опций; `long_options` — описание

длинных опций; `optlist` — список двухэлементных кортежей (опция, значение); `args` — чистые аргументы командной строки.

Описание однобуквенных опций — строка из букв используемых опций. Опции, которые требуют значения, обозначаются буквой и последующим двоеточием ":". Описание длинных опций — список имен этих опций (без "--"). После имен длинных опций, требующих значения, ставится "=".

Замечание

После специальной опции "--" все остальные аргументы командной строки считаются аргументами, а не опциями. Это придумано для того, чтобы программам можно было задавать файлы, имена которых начинаются с дефиса.

Пример в листинге 9.32 и результат исполнения программы с по-разному заданными опциями помогут быстрее разобраться с функцией `getopt()`:

Листинг 9.32. Файл getopt_eg1.py

```
#!/usr/bin/python
import sys
from getopt import getopt
(optlist, args) = getopt(sys.argv[1:], "ab:", ["help", "version"])
print "optlist:", optlist
print "args:", args
```

Результат выполнения:

```
python getopt_eg1.py 1 2 3
optlist: []
args: ['1', '2', '3']
python getopt_eg1.py -a -b 1 2 3
optlist: [('-a', ''), ('-b', '1')]
args: ['2', '3']
python getopt_eg1.py -- -a -b 1 2 3
optlist: []
args: ['-a', '-b', '1', '2', '3']
python getopt_eg1.py --help --version 1 2 3
optlist: [('--help', ''), ('--version', '')]
args: ['1', '2', '3']
python getopt_eg1.py --help --version -a -bopt 1 2 3
optlist: [('--help', ''), ('--version', ''), ('-a', ''), ('-b', 'opt')]
args: ['1', '2', '3']
```

```
python getopt_eg1.py -b opt -b opt1 1 2 3
optlist: [('-b', 'opt'), ('-b', 'opt1')]
args: ['1', '2', '3']
python getopt_eg1.py -a 1 -b 2
optlist: [('-a', '')]
args: ['1', '-b', '2']
python getopt_eg1.py -a -b 2 -b 3
optlist: [('-a', ''), ('-b', '2'), ('-b', '3')]
args: []
```

Разумеется, в случае возникновения ошибок возбуждается исключение `getopt.GetoptError`:

```
python2.1 getopt_eg1.py -c
Traceback (most recent call last):
  File "getopt_eg1.py", line 12, in ?
    (optlist, args) = getopt(sys.argv[1:], "ab:", ["help", "version"])
...
getopt.GetoptError: option -c not recognized
```

9.7.6. Модуль *glob*

С помощью модуля `glob` можно получать список путей к файлам, соответствующих заданному шаблону. В шаблоне могут присутствовать следующие специальные символы:

- `"*"` — любое число (возможно, ноль) любых символов;
- `"?"` — любой символ;
- `"[..."]"` — любой символ из заданного набора и/или диапазона.

Следующие примеры были запущены в каталоге со стандартными модулями Python:

```
>>> from glob import glob
>>> print glob("/usr/lib/python2.1/U*.py")
['/usr/lib/python2.1/UserDict.py', '/usr/lib/python2.1/UserList.py',
 '/usr/lib/python2.1/UserString.py']
>>> print glob("/usr/lib/python2.1/U*.py?")
['/usr/lib/python2.1/UserDict.pyc', '/usr/lib/python2.1/UserDict.pyo',
 '/usr/lib/python2.1/UserList.pyc', '/usr/lib/python2.1/UserList.pyo',
 '/usr/lib/python2.1/UserString.pyc',
 '/usr/lib/python2.1/UserString.pyo']
```

```
>>> print glob("/usr/lib/python2.1/*[hH][tT][tT][pP]*.py")
['/usr/lib/python2.1/BaseHTTPServer.py',
 '/usr/lib/python2.1/CGIHTTPServer.py',
 '/usr/lib/python2.1/SimpleHTTPServer.py',
 '/usr/lib/python2.1/httplib.py']
```

В первом случае мы получаем список файлов, чьи имена начинаются на "U" и оканчиваются на ".py". Во втором случае выбираемые имена тоже начинаются на "U", но в качестве окончания выбрано ".py" и любой символ. В результате выбираются компилированные файлы ".pyc" и оптимизированные компилированные файлы ".pyo". В третьем случае мы выбираем все .py-файлы, в которых присутствует http (в любом сочетании строчных и прописных букв).

Заметьте также, что "*" может встречаться в любом месте пути, а не только в имени файла.

9.7.7. Модуль *popen2*

Модуль popen2 содержит функции для запуска дочерних процессов, связанных с родительским несколькими каналами ввода/вывода. Если функция os.popen() позволяла порождать процесс, связанный всего одним каналом с породившим его процессом, то popen2 позволяет контролировать все три стандартных канала: stdin, stdout и stderr.

```
popen2(cmd[, bufsize[, mode]]) -> (ch_stdout, ch_stdin)
popen3(cmd[, bufsize[, mode]]) -> (ch_stdout, ch_stdin, ch_stderr)
popen4(cmd[, bufsize[, mode]]) -> (ch_stdout_stderr, ch_stdin)
```

Вызывает команду cmd. Если указан bufsize, при вводе/выводе будет использоваться буфер указанного размера. Режим mode может быть t (текстовым) или b (бинарным). Вывод из дочернего процесса осуществляется по каналам ch_stdout (стандартный вывод), ch_stderr (стандартный вывод ошибок), ch_stdout_stderr (стандартный вывод и вывод ошибок вместе), а ввод осуществляется через канал ch_stdin (стандартный ввод). В данном случае ввод и вывод обозначаются с точки зрения дочернего процесса. Родительский процесс может писать в ch_stdin и читать из других каналов ввода/вывода.

В следующем примере мы используем утилиту tac (имеющуюся в некоторых UNIX-системах) для получения записываемых строк в обратном порядке.

```
#!/usr/bin/python
from popen2 import popen2
(from_tac, to_tac) = popen2("tac")
```

```
for i in range(1, 6):
  to_tac.write(str(i)+"\n")
to_tac.close()
print from_tac.read()
from_tac.close()
```

Результат выполнения:

```
5
4
3
2
1
```

9.7.8. Модуль *shutil*

Модуль `shutil` содержит функции для копирования файлов и каталогов, их свойств, а также для копирования и удаления дерева каталогов.

- `shutil.copy(src, dst)`

  Копирует файл `src` в `dst` (может быть каталогом). Аналог команды `cp src dst` в UNIX.

- `shutil.copy2(src, dst)`

  Копирует файл `src` в `dst` (может быть каталогом) вместе со статусной информацией. Аналог команды `cp -p src dst` в UNIX.

- `shutil.copyfile(src, dst)`

  Копирует данные из файла `src` в файл `dst`.

- `shutil.copymode(src, dst)`

  Устанавливает режим файла `dst` как у `src`.

- `shutil.copystat(src, dst)`

  Устанавливает статусную информацию (режим, время последней модификации и доступа) файла `dst` как у `src`.

- `shutil.copytree(src, dst[, symlinks])`

  Копирует дерево каталогов, используя `copy2()`. Каталог `dst` должен отсутствовать. Если флаг `symlinks` — истина, символические ссылки копируются как символические ссылки. В противном случае копируется содержимое, на которое указывают эти ссылки. Ошибки при копировании выводятся на стандартный вывод. Эту функцию лучше рассматривать как пример, чем надежное средство для копирования дерева каталогов.

- `rmtree(path[, ignore_errors=0][, onerror=None])`

  Удаляет дерево каталогов `path`. Если `ignore_errors` — истина, ошибки игнорируются. Если задана функция `onerror()`, она вызывается для обработки ошибки. Функция `onerror()` получает три параметра: `cmd_func` — функция, при выполнении которой произошла ошибка; `path` — путь, при удалении которого произошла ошибка и `exc` — информация об исключении в виде, возвращаемом функцией `sys.exc_info()`.

9.7.9. Модуль *select*

Модуль `select` предоставляет доступ к функциям `select()` и `poll()` для ожидания завершения ввода/вывода.

- `select.select(rl, wl, xl[, timeout]) -> (rl, wl, xl)`

  где `rl` — список объектов от которых ожидается готовность по чтению; `wl` — то же, по записи; `xl` — то же, по появлению нештатных ситуаций; `timeout` — значение тайм-аута в секундах (число с плавающей точкой).

  Функция `select()` ждет изменения состояния одного или более файловых дескрипторов (файлов или сокетов). Возвращает кортеж со списками файловых дескрипторов с соответствующими состояниями. Если функция завершилась по тайм-ауту, эти списки могут быть пустыми. Если в качестве тайм-аута указан `0`, функция возвращает результат сразу. В противном случае функция ждет, пока хотя бы один файловый объект будет в соответствующем состоянии. В Windows в вызове `select()` можно использовать только файловые дескрипторы сокетов.

- `select.poll(...) -> pobj`

  Возвращает объект для выполнения опроса `pobj`, который позволяет регистрировать файловые дескрипторы для опроса событий ввода/вывода. Методы poll-объектов представлены в табл. 9.42. Файловый дескриптор `fd` в этой таблице понимается не только как целое число (собственно файловый дескриптор), но и как любой файловый объект, имеющий метод `fileno()`.

Таблица 9.42. *Методы `poll`-объектов*

Метод	Описание
`register (fd[, mask])`	Регистрирует файловый дескриптор для опроса определенных `mask` событий. Параметр `mask` может быть составлен побитовым ИЛИ констант (табл. 9.43). По умолчанию опрашиваются все три события: `POLLIN`, `POLLOUT`, `POLLPRI`. Повторная регистрация ошибки не вызывает

Таблица 9.42 *(окончание)*

Метод	Описание
`unregister(fd)`	Убрать файловый дескриптор из списка отслеживаемых `poll`-объектом. При попытке снять с учета файловый дескриптор, который не был зарегистрирован, возбуждается исключение `KeyError`
`poll([timeout]) -> lst`	Опрашивает зарегистрированные файловые дескрипторы и возвращает (возможно, пустой) список кортежей `(fd, event)`, где `fd` — файловый дескриптор, а `event` — ситуация, в которой он находится. Для анализа ситуации `event` можно применять побитовое И с соответствующей константой

Таблица 9.43. *Константы из модуля `select` для описания опрашиваемых ситуаций*

Константа	Значение
`POLLIN`	Есть данные для чтения
`POLLPRI`	Есть срочные данные для чтения
`POLLOUT`	Готовность к выводу
`POLLERR`	Ошибка
`POLLHUP`	Зависание (hang up)
`POLLNVAL`	Неверный запрос

Программа в листинге 9.33 ожидает ввода из `stdin`, выводя каждую секунду символы `"?"`. Если ввод не наступает после 5 секунд, она выводит фразу `"Время вышло!"`.

Листинг 9.33. Файл fastinput.py. Нетерпеливый ввод

```
import select, sys
while 1:
  print "5 секунд на ввод:"
  s = 0
  while s < 5:
    s = s+1
    p = select.select([sys.stdin], [], [], 1)
    if p[0]:
      break
    sys.stderr.write("?")
  if p[0]:
```

```
    data = raw_input()
    print "\nВведено: ", data
  else:
    print "\nВремя вышло!"
    break
```

Результат выполнения:

```
5 секунд на ввод:
???1

Введено:  1
5 секунд на ввод:
?????
Время вышло!
```

9.7.10. Модуль *signal*

Во время работы программы в операционной системе могут возникать некоторые события, о которых операционная система сообщает программе с помощью *сигналов*. Модуль `signal` предоставляет возможность обрабатывать сигналы путем задания обработчиков для каждого вида сигнала. В следующем примере мы определяем обработчик для обработки прерывания программы с клавиатуры по нажатии комбпнации клавиш <Ctrl>+<C> (<Ctrl>+<Break>):

Листинг 9.34. Обработка прерывания программы с клавиатуры

```
#!/usr/bin/python
import signal, time
# Функция обработчика
def kbd_handler(signum, frame):
    print "Ой!"
# Регистрация обработчика сигнала SIGINT
signal.signal(signal.SIGINT, kbd_handler)
# Небольшой цикл с задержкой
for i in range(100):
    time.sleep(10)
```

Эту программу нужно запустить из командной строки и попробовать ее прервать с клавиатуры. Кстати, здесь же можно видеть особенность функции `time.sleep()` — он не продолжает "спать" после обработки прерывания.

Стандартные (POSIX.1) сигналы приведены в табл. 9.44, а сигналы, характерные для других систем, в табл. 9.45. Обозначения, используемые в графе

"Действие по умолчанию", расшифровываются в табл. 9.46. Эта графа определяет действия ОС в случае, когда сигнал не обработан программой, и другие особенности сигнала.

Таблица 9.44. *Некоторые сигналы, определенные в POSIX.1*

Сигнал	Номер сигнала	Действие по умолчанию	Описание
SIGHUP	1	A	Убит контролирующий процесс
SIGINT	2	A	Прерывание с клавиатуры
SIGQUIT	3	C	Выход с клавиатуры
SIGILL	4	C	Некорректная операция
SIGABRT	6	C	Сигнал прерывания abort
SIGFPE	8	C	Исключение операции с плавающей точкой
SIGKILL	9	AEF	Безусловно убивает процесс
SIGSEGV	11	C	Ошибка обращения к памяти
SIGPIPE	13	A	Обрыв канала (запись в канал, откуда никто не читает)
SIGALRM	14	A	Сигнал таймера ("будильник")
SIGTERM	15	A	Сигнал завершения процесса
SIGUSR1	10	A	Определяемый пользователем сигнал 1
SIGUSR2	12	A	Определяемый пользователем сигнал 2
SIGCHLD	17	B	Завершение или остановка дочернего процесса
SIGCONT	18		Возобновление после остановки
SIGSTOP	19	DEF	Остановка процесса
SIGTSTP	20	D	Остановка с телетайпного устройства (tty)
SIGTTIN	21	D	tty-ввод для фонового процесса
SIGTTOU	22	D	tty-вывод для фонового процесса

Таблица 9.45. *Другие сигналы, доступные в различных системах*

Сигнал	Номер сигнала	Действие по умолчанию	Описание
SIGBUS	7	C	Ошибка шины
SIGINFO	29	A	Синоним SIGPWR

Таблица 9.45 *(окончание)*

Сигнал	Номер сигнала	Действие по умолчанию	Описание
SIGIO	29	A	Ввод/вывод возможен
SIGIOT	6	C	Ловушка ввода/вывода. Синоним SIGABRT
SIGPOLL	29	A	Опрашиваемое (pollable) событие. Синоним SIGIO
SIGPROF	27	A	Таймер профайлинга
SIGPWR	30	A	Сбой по электропитанию
SIGSTKFLT	16	A	Сбой стека на сопроцессоре
SIGTRAP	5	C	Ловушка трассировки или точка останова
SIGUNUSED	31	A	Неиспользуемый сигнал
SIGURG	23	B	Сокет требует вмешательства
SIGVTALRM	26	A	Виртуальный "будильник"
SIGWINCH	28	B	Изменение размеров окна
SIGXCPU	24	C	Вышел предел использования времени процессора
SIGXFSZ	25	C	Превышен размер файла

Примечание

Номера некоторых сигналов в разных операционных системах могут различаться. Более того, некоторые сигналы могут вообще не работать. Модуль signal правильно определяет соответствующие константы.

Таблица 9.46. *Действия операционной системы*

Условное обозначение	Обозначаемое действие
A	Завершение процесса
B	Игнорирование сигнала
C	Завершение процесса с записью дампа памяти (core файла)
D	Остановка процесса
E	Сигнал не может быть пойман (программой)
F	Сигнал не может быть проигнорирован

В документации сделаны следующие замечания относительно применения сигналов в Python:

- Обработчик сигнала действует в данной программе после регистрации функцией `signal.signal()` до тех пор, пока не будет заменен другим или отменен.
- Нельзя оградить фрагмент кода от действия сигналов.
- Обработчики сигналов вызываются асинхронно (с точки зрения пользователя Python) между элементарными инструкциями интерпретатора Python, поэтому обработка сигналов может задержаться на произвольно долгое время.
- При поступлении сигнала в момент выполнения операции ввода/вывода последняя может возбудить исключение после выполнения обработчика сигнала.
- В Python нет смысла ловить сигналы о синхронных ошибках вроде SIGFPE и SIGSEGV.
- Python ловит некоторые сигналы сам. Сигнал SIGPIPE игнорируется (ошибки, связанные с каналами и сокетами, возбуждают обычные исключения Python). SIGINT возбуждает исключение `KeyboardInterrupt`. Эти сигналы могут быть переопределены.
- При использовании в одной программе и сигналов, и потоков (threads) нужно соблюдать некоторые дополнительные правила. Во-первых, регистрировать обработчики сигналов можно только в главном потоке. Во-вторых, обрабатываться сигналы будут тоже в главном потоке. В любом потоке могут выполняться `signal.alarm()`, `signal.getsignal()` и `signal.pause()`. То есть сигналы нельзя использовать для коммуникации между потоками *(см. разд. 9.7.13)*.

Помимо приведенных в табл. 9.44 и 9.45 символических имен сигналов, в модуле `signal` определены следующие функции и переменные:

- `signal.signal(signalnum, handler) -> prev_handler`

  где `signalnum` — номер сигнала, например, `signal.SIGPIPE`; `handler` — функция обработчика сигнала или одно из специальных значений: `signal.SIG_IGN` (для игнорирования сигнала) и `signal.SIG_DFL` (для обработки сигнала по принятому в Python умолчанию); `prev_handler` — предыдущий обработчик сигнала.

  Функция обработчика сигнала `handler()` должна принимать два аргумента. Первый аргумент получает номер сигнала, а второй — фрейм-объект *(см. статью "Фрейм-объекты" в разд. 8.2.10)* с текущим фреймом исполнения.

- `signal.alarm(secs) -> prev_secs`

  Функция `signal.alarm()` устанавливает таймер для вызова SIGALRM через `secs` секунд. Ранее запланированный сигнал отменяется. Возвращается

количество секунд до срабатывания ранее установленного таймера (или 0, если таймер не был установлен). Если `secs` задан равным нулю, таймер не устанавливается, а установленный ранее таймер сбрасывается.

- `signal.getsignal(signalnum) -> handler`

  Функция `signal.getsignal()` возвращает текущий обработчик сигнала `signalnum` (Python-объект с поддержкой вызова), либо одно из специальных значений: `signal.SIG_IGN` (сигнал игнорируется), `signal.SIG_DFL` (сигнал обрабатывается обработчиком по умолчанию) или `None` (если Python не обрабатывает данный сигнал).

- Функция `signal.pause()` усыпляет процесс до получения какого-либо сигнала, после чего будет вызван соответствующий обработчик.

9.7.11. Модуль *stat*

Функции и константы модуля `stat` помогают расшифровывать результаты выполнения функций `os.stat()`, `os.fstat()`, `os.lstat()`. Эти функции возвращают кортежи из (как минимум) десяти значений. В табл. 9.47 приведены константы для доступа к определенным элементам кортежа.

Таблица 9.47. *Индексы для доступа к элементам кортежа с данными о файле*

Индекс	Константа	Значение поля кортежа
0	`stat.ST_MODE`	Режим файла (правда доступа, тип файла и т. п.)
1	`stat.ST_INO`	Номер индексного узла, соответствующего файлу, в файловой системе
2	`stat.ST_DEV`	Устройство, на котором находится файл
3	`stat.ST_NLINK`	Количество жестких ссылок на файл
4	`stat.ST_UID`	Идентификатор пользователя — владельца файла
5	`stat.ST_GID`	Идентификатор группы владельца файла
6	`stat.ST_SIZE`	Размер файла в байтах (для обычных файлов) или количество байтов в очереди для чтения (для некоторых других видов файлов)
7	`stat.ST_ATIME`	Время последнего обращения к содержимому файла (работает не для всех файловых систем)
8	`stat.ST_MTIME`	Время последней модификации содержимого файла
9	`stat.ST_CTIME`	Время последнего изменения статуса файла (то есть информации в индексном узле файла)
>10		Некоторые платформы могут давать дополнительные элементы

Примечание

В табл. 9.47 то, что мы называем файлами, относится к *индексным узлам (inode)* в UNIX-системах. Индексные узлы хранят информацию о файле или каталоге, часть которой мы и получаем в виде десятиэлементного кортежа. В системе UNIX несколько файлов могут иметь один и тот же inode, поэтому `stat.ST_NLINK` показывает количество файлов, связанных с данным индексным узлом.

Следующий пример показывает, как можно проверить, как давно файл был изменен:

```
import os, time
from stat import *
file = "file.dat"
delta = time.time() - os.stat(file)[ST_MTIME]
print "Файл %(file)s был изменен %(delta)i секунд назад." % vars()
```

Результат выполнения:

```
Файл file.dat был изменен 1473096 секунд назад.
```

В табл. 9.48 приведены функции модуля `stat`, которые на основании режима файла (`ST_MODE`, нулевой элемент кортежа) определяют, принадлежит ли файл к определенному типу, возвращая ненулевое значение.

Таблица 9.48. *Определение типа файла*

Функция	Какой тип файла определяет
`stat.S_ISDIR(mode)`	Каталог
`stat.S_ISCHR(mode)`	Символьное устройство
`stat.S_ISBLK(mode)`	Блочное устройство
`stat.S_ISREG(mode)`	Обычный файл
`stat.S_ISFIFO(mode)`	Именованный канал (pipe)
`stat.S_ISLNK(mode)`	Символическая ссылка
`stat.S_ISSOCK(mode)`	Сокет

Кроме того, функция `stat.S_IMODE()` выделяет часть значения `mode`, соответствующую правам доступа, изменяемым с помощью функции `os.chmod()`, а `S_IFMT(mode)` выделяет из `mode` часть битов, которые используются функциями определения типа файла, приведенными выше.

Примечание

Для определения типа файла лучше использовать соответствующие функции модуля `os.path`. Приведенные в этом разделе функции выгодно использовать в случае, когда проверяется сразу несколько параметров, относящихся к файлу. То есть можно узнать нужную информацию с помощью одного вызова `stat()`.

Значение `mode` представляет собой целое число, каждый бит которого отвечает за определенный признак. Взять определенный бит из значения `mode` можно путем побитового И с маской, в которой установлен только этот бит. В табл. 9.49 и 9.50 приведены определенные в модуле `stat` маски для доступа к битам (или группам битов) значения `mode`.

Таблица 9.49. *Маски для выделения значений отдельных битов режима файла*

Символическое имя	Значение маски	Описание
`stat.S_IXOTH`	1	Исполнение файла (просмотр каталога) разрешено всем
`stat.S_IWOTH`	2	Запись в файл разрешена всем
`stat.S_IROTH`	4	Чтение из файла разрешено всем
`stat.S_IRWXO`	7	Исполнение (просмотр), запись или чтение разрешено для всех
`stat.S_IXGRP`	8	Исполнение файла (просмотр каталога) разрешено группе
`stat.S_IWGRP`	16	Запись в файл разрешена группе
`stat.S_IRGRP`	32	Чтение из файла разрешено группе
`stat.S_IRWXG`	56	Исполнение (просмотр), запись или чтение разрешено группе
`stat.S_IEXEC`, `stat.S_IXUSR`	64	Исполнение файла (просмотр каталога) разрешено владельцу
`stat.S_IWRITE`, `stat.S_IWUSR`	128	Запись в файл разрешена владельцу
`stat.S_IREAD`, `stat.S_IRUSR`	256	Исполнение файла (просмотр каталога) разрешено владельцу
`stat.S_IRWXU`	448	Исполнение (просмотр), запись или чтение разрешено владельцу

О работе `stat.S_ISVTX`, `stat.S_ISGID` и `stat.S_ISUID` лучше справиться в документации к конкретной операционной системе.

Таблица 9.50. *Маски для определения типа файла*

Символическое имя	Значение маски	Тип файла
stat.S_IFIFO	4096	Канал (pipe)
stat.S_IFCHR	8192	Символьное устройство
stat.S_IFDIR	16384	Каталог
stat.S_IFBLK	24576	Блочное устройство
stat.S_IFREG	32768	Обычный файл
stat.S_IFLNK	40960	Символическая ссылка
stat.S_IFSOCK	49152	Сокет

Следующий пример показывает, как прочитать права на чтение для конкретного файла:

```
import os
from stat import *
file = "/etc/passwd"
mode = os.stat(file)[ST_MODE]
if mode & S_IROTH:
  print "Файл %s может читать любой пользователь" % file
else:
  print "Файл %s не может читать любой пользователь" % file
if mode & S_IWOTH:
  print "Файл %s может изменять любой пользователь" % file
else:
  print "Файл %s не может изменять любой пользователь" % file
```

Результат выполнения:

```
Файл /etc/passwd может читать любой пользователь
Файл /etc/passwd не может изменять любой пользователь
```

9.7.12. Модуль *tempfile*

Модуль `tempfile` служит для формирования имени и открытия временного файла в каталоге для временных файлов.

- `tempfile.mktemp([suffix]) -> string`

  где `suffix` — необязательный параметр, обозначающий суффикс, добавляемый к концу сгенерированного имени временного файла.

Функция `tempfile.mktemp()` генерирует уникальное имя для временного файла. Можно задать суффикс, на который это имя будет оканчиваться. Функция возвращает абсолютный путь к файлу.

- `tempfile.TemporaryFile([mode[, bufsize[, suffix]]]) -> file`

  где `mode` — режим открытия файла, по умолчанию `"w+b"`; `bufsize` — размер буфера; `suffix` — суффикс, присоединяемый к концу имени файла; `file` — файловый объект.

  Если в программе требуется некоторый временный файл, который можно читать и в который одновременно можно записывать, то файл можно создать с помощью `tempfile.TemporaryFile()`:

```
>>> from tempfile import TemporaryFile
>>> tmpfile = TemporaryFile()
>>> tmpfile.write("Привет")
>>> tmpfile.seek(0)
>>> print tmpfile.read()
Привет
```

Временный файл не нужно закрывать, т. к. он сам закроется (и уничтожится) при удалении соответствующего ему объекта.

9.7.13. Модуль *threading*

В *статье "Создание и управление процессами" разд. 9.7.1* мы говорили об одном из вариантов реализации параллельного исполнения кода — использовании процессов. Здесь мы рассмотрим другой подход, основанный на *потоках управления (threads)*. (Слово *thread* в переводе с англ. означает нить.)

Потоки управления образуются и живут в рамках одного процесса. В однопоточном приложении (программе, не использующей дополнительных потоков) есть только один поток управления: при запуске программы он последовательно исполняет встреченные в программе операторы, направляется по одной из альтернативных ветвей операторов выбора, несколько раз проходит через тело цикла, выбирается наверх при возникновении исключений и т. п. В любой момент времени интерпретатор Python знает, какую команду исполнить, а после исполнения команды — куда передать управление. Эта ниточка непрерывна все время исполнения программы и обрывается только по завершении программы.

Представим теперь, что в некоторой точке ниточка раздваивается и каждый поток пошел своим путем, возможно, еще несколько раз раздваиваясь по пути. (При этом один из потоков всегда остается главным, и его завершение означает завершение всей программы.) В каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет внимание каждому потоку. Однако такое незначительное усложнение ис-

полнения программы приводит к тому, что нам необходимы механизмы для согласования деятельности потоков. Например, нельзя допускать, чтобы потоки одновременно изменяли один и тот же объект, т. к. результат такого действия скорее всего не будет корректным. На помощь приходят специальные объекты, называемые *семафорами*. Семафоры позволяют исключать исполнение одного и того же участка кода несколькими потоками одновременно. Самый простой двоичный семафор — *замок (lock)* или mutex (от англ. *mutually exclusive* — взаимоисключающий). Чтобы поток мог продолжить исполнение кода, он должен сначала овладеть замком. После этого он выполняет некоторый участок кода и снимает замок, чтобы другой поток (возможно, уже сделавший запрос на данный замок) мог его получить и пройти дальше к выполнению охраняемого замком участка программы.

Для создания и управления потоками применяются объекты класса `Thread`. В модуле `threading` также представлены необходимые для работы *многопоточной (multithreading) программы* аксессуары: несколько видов семафоров (объекты `Lock`, `RLock`, `Semaphore`), других механизмов коммуникации между потоками (объекты `Event`, `Condition`).

Класс *threading.Thread*

Класс `threading.Thread` позволяет назначить действия, которые должны выполняться в отдельном потоке, и имеет следующий конструктор.

```
threading.Thread(group, target, name, args=(), kwargs={}) -> obj
```

где `group` — группа потоков (пока не реализовано, значение должно быть `None`); `target` — функция, метод или другой объект, позволяющий вызов (по умолчанию `None`), именно он вызывается при запуске потока; `name` — имя потока (может быть не задано или задано как `None`, в этом случае формируется автоматически); `args` — позиционные аргументы для вызова потока (по умолчанию `()`); `kwargs` — именованные аргументы для вызова потока (по умолчанию `{}`).

Имеется и другой способ задания `target` — наследование от класса `threading.Thread` и переопределение метода `run()` (другие методы переопределять нельзя). Методы этого класса описаны в табл. 9.51.

Таблица 9.51. *Методы объектов класса* `threading.Thread`

Метод	Описание
`start()`	Запускает поток
`run()`	Метод для вызова в отдельном потоке. Получает свое значение из параметра `target` конструктора класса или переопределением при наследовании от `threading.Thread`. При вызове используются `args` и `kwargs`, заданные в конструкторе

Таблица 9.51 *(окончание)*

Метод	Описание
`join([timeout])`	Ожидает завершения потока. Поток, который вызывает этот метод, приостанавливается. Задание `timeout` (число с плавающей точкой) позволяет задать время ожидания (в секундах), после которого приостановленный поток продолжает свою работу независимо от того, завершился ли поток, метод `join()` которого был вызван. Поток не может вызвать метод `join()` от самого себя, т. к. это заводит его в тупик. Также нельзя ожидать завершения незапущенного еще потока
`getName()`	Возвращает имя потока. Для главного потока это `"MainThread"`
`setName(name)`	Устанавливает имя потока
`isAlive()`	Возвращает значение "истина", если поток работает (то есть метод `run()` уже вызван, но еще не завершился)
`isDaemon()`	Возвращает значение "истина", если поток имеет признак демона. Программа на Python считается завершенной, если завершились все потоки, не являющиеся демонами
`setDaemon(flag)`	Устанавливает признак того, что поток является демоном. Начальное значение этого признака заимствуется от потока, запустившего данный поток. Признак можно устанавливать только для незапущенных еще потоков

Класс *threading.Lock*

Объект класса `Lock` — простейший замок, реализованный на основе модуля `thread` *(см. разд. 9.7.14)*. Замок имеет два состояния (он может быть либо открыт, либо заперт). Объект `Lock` имеет два основных метода `acquire()` (с помощью этого метода поток делает запрос на запирание замка) и `release()` (для снятия замка). Сразу после создания замок находится в открытом состоянии. В этом состоянии первый запрос методом `acquire` приводит замок в запертое состояние. Все остальные потоки блокируются при вызове `acquire()` вплоть до снятия замка, после чего один из них его закрывает и возвращается из `acquire()`.

С помощью этого вида замка можно обеспечить выполнение некоторого участка программы одновременно только одним потоком. Следует иметь в виду, что еще раз запросить закрытый замок не может даже сам процесс, который его до этого запер: это приводит к бесконечному ожиданию. Однако, отпереть замок может любой процесс.

Методы объектов класса `Lock` описаны в *разд. 9.7.14*.

Замки очень часто ассоциируются с некоторыми неразделяемыми между потоками ресурсами для защиты этих ресурсов от одновременного исполь-

зования. Однако в программе таких и ресурсов может быть несколько и поэтому при работе с замками очень важно хорошо продумать, не зайдет ли выполнение программы в тупик из-за того, что двум потокам потребуются одни и те же ресурсы, но ни тот, ни другой не могут их получить, т. к. уже получили некоторые замки. Такая ситуация называется *тупиком (deadlock)* и проиллюстрирована листингом 9.35.

Листинг 9.35. Файл deadlock.py. Тупиковая ситуация

```
import threading, time

resource_A = threading.Lock()
resource_B = threading.Lock()

def process1():
  print "процесс 1 запрашивает ресурс A"
  resource_A.acquire()
  print "процесс 1 получил ресурс A"
  time.sleep(1)
  print "процесс 1 запрашивает ресурс B"
  resource_B.acquire()
  print "процесс 1 получил ресурс B"
  resource_A.release()
  resource_B.release()

def process2():
  print "процесс 2 запрашивает ресурс B"
  resource_B.acquire()
  print "процесс 2 получил ресурс B"
  time.sleep(1)
  print "процесс 2 запрашивает ресурс A"
  resource_A.acquire()
  print "процесс 2 получил ресурс A"
  resource_A.release()
  resource_B.release()

threading.Thread(target=process1).start()
process2()
print "Тупик..."
```

Результат выполнения:

```
процесс 2 запрашивает ресурс B
процесс 2 получил ресурс B
процесс 1 запрашивает ресурс A
процесс 1 получил ресурс A
процесс 2 запрашивает ресурс A
процесс 1 запрашивает ресурс B
```

Разумеется, возникает резонный вопрос: как избежать подобных ситуаций? Скорее всего, придется ввести некоторые дополнительные правила. Например, запрашивать ресурсы всегда в определенном порядке.

Класс *threading.RLock*

В некоторых случаях (например, в рекурсивных функциях) необходимо, чтобы один и тот же поток имел возможность запрашивать замок даже в случае, если он им уже обладает. Объект класса `RLock` имеет эту возможность и потому поток может сколько угодно раз запрашивать замок методом `acquire()` и столько же раз снимать методом `release()`. Первоначально замок находится в открытом состоянии и не принадлежит ни одному потоку. Замок возвращается в исходное незапертое состояние только после одинакового числа запросов и освобождений. При этом все остальные потоки, которые запрашивают замок, остаются в состоянии ожидания.

Семафоры: класс *threading.Semaphore*

Семафоры или как их иногда называют — семафоры Дейкстры — по имени их изобретателя, предоставляют более общий механизм синхронизации потоков, чем замки. С помощью семафоров в критическую часть программы допускаются несколько потоков (в частном случае — всего один). Семафор ведет счетчик запросов, который при каждом запросе `acquire()` уменьшается на единицу, а при каждом `release()` — увеличивается на единицу. Счетчик не может стать меньше нуля, поэтому, если запрос поступает когда счетчик равен нулю, потоку (или потокам) приходится ждать, как и в случае с замками, пока один из потоков не увеличит счетчик.

Конструктор класса `Semaphore` принимает единственный необязательный аргумент — начальное состояние счетчика (по умолчанию оно равно `1`, что соответствует замку класса `Lock`). Методы `acquire()` и `release()` действуют аналогично таким же методам у замков, описанных выше.

Класс *threading.Event*

Объекты класса `threading.Event` (в переводе с английского — событие) служат для простейшей коммуникации между потоками, при которой один

поток сигнализирует о событии, тогда как один или более других потоков находятся в состоянии ожидания. Объекты-события имеют внутренний флаг, который может быть установлен или сброшен. При своем создании флаг события находится в сброшенном состоянии. Если флаг установлен, ожидания не происходит: соответствующий поток просто продолжает свою работу. В табл. 9.52 описаны методы объектов класса `threading.Event`.

Таблица 9.52. *Методы объектов-событий*

Метод	Описание
`set()()`	Устанавливает внутренний флаг. При этом все ждущие данного события потоки выходят из состояния ожидания
`clear()()`	Сбрасывает флаг. Все события, которые вызывают `wait()`, будут находиться в состоянии ожидания до тех пора, пока флаг сброшен или истек тайм-аут
`isSet()()`	Возвращает значение "истина" если флаг установлен
`wait([timeout])()`	Переводит поток в состояние ожидания, если флаг сброшен, и сразу возвращается, если он установлен. Если был задан аргумент `timeout` (в секундах) и по истечении этого времени событие не наступило, метод также возвращается

Шуточный пример ниже иллюстрирует применение объектов-событий:

Листинг 9.36. Файл pioneers.py. Пример коммуникации между потоками

```
import threading
ready = threading.Event()

def pioneer():
  ready.wait()
  print "Всегда готовы!"

for i in range(5):
  p = threading.Thread(target=pioneer)
  p.start()

print "Пионеры, будьте готовы!"
ready.set()
```

Класс *Condition*

Экземпляры класса `Condition` (условие) являются более сложным методом коммуникации между потоками. Объект-условие служит механизмом для оповещения потоков о том, что наступило изменение некоторого состояния и они могут проверить ситуацию.

Конструктор класса `Condition` принимает необязательный аргумент, в котором можно указать объект-замок класса `Lock` или `RLock`. Если аргумент не задан, создается новый замок класса `Rlock` (табл. 9.53).

Таблица 9.53. *Методы класса* `Condition`

Метод	Описание
`acquire(...)`	Запрашивает замок. Фактически вызывается одноименный метод принадлежащего объекту-условию объекта-замка
`release()`	Снимает замок
`wait([timeout])`	Поток переходит в режим ожидания. Этот метод может быть вызван только если вызывающий его поток получил замок. Метод снимает замок и блокирует поток до соответствующих объявлений, т. е. вызовов методов `notify()` и `notifyAll()` другими потоками. Как обычно, необязательный аргумент `timeout` задает количество секунд, которые поток будет ждать. При выходе из ожидания поток снова запрашивает замок и возвращается из метода `wait()`. Если замок, использованный в условии, основан на `RLock`, метод `wait` использует знание его внутреннего устройства с тем, чтобы обеспечить снятие замка на время ожидания независимо от состояния его внутреннего счетчика. После возвращения из `wait()`, тем же путем восстанавливается исходное значение счетчика
`notify()`	Выводит из режима ожидания любой из потоков, ожидающих данное условие. Метод можно вызывать только получив замок, ассоциированный с условием. Документация предупреждает, что в будущих реализациях модуля из целей оптимизации этот метод будет прерывать ожидание сразу нескольких потоков. Сам по себе метод `notify()` не приводит к продолжению исполнения ожидавших условия потоков, т. к. этому мешает занятый замок. Потоки получают управление только после снятия замка тем же потоком, что вызвал `notify()`
`notifyAll`	Метод, аналогичный `notify`, но прерывает ожидание всех ждущих события потоков

В листинге 9.37 показан пример использования объектов класса `Condition`. Программа выдает приглашение `"вводите:"` и позволяет ввести одно или несколько значений, разделенных `";"` — целых чисел от 0 до 999. Для этих зна-

чений считаются факториалы и выводится количество цифр в результате. Для достаточно больших чисел вычисления могут быть выполнены не сразу, и потому приглашение ко вводу появляется тут сразу. Вычисления производят два потока `ev1` и `ev2`, которые оповещаются о том, что список исходных данных `to_eval` имеет хотя бы одно значение. Для удобства наблюдения каждый поток предваряет вывод своим позывным: `главный`, `1` или `2`. Конечно, пример является достаточно искусственным для однопроцессорного компьютера.

Листинг 9.37. Файл Threaded_Eval.py. Распараллеливание вычислений

```
import threading, string, sys, time

cond = threading.Condition()

def fact(n):
  if n <= 1:
    return 1L
  else:
    return fact(n-1)*n

def evaluator(name):
  while 1:
    cond.acquire()
    while not to_eval:
      print "%s ждет:" % name
      cond.wait()
      print "%s активен:" % name
    v = to_eval[0]
    del to_eval[0]
    cond.release()
    print "%s выч: len(%s!) = %s" % (name, v, len(str(fact(int(v)))))

to_eval = []
ev1 = threading.Thread(target=evaluator, args=("1",))
ev1.setDaemon(1)
ev1.start()
ev2 = threading.Thread(target=evaluator, args=("2",))
ev2.setDaemon(1)
ev2.start()
```

```
while 1:
  print "главный вводите:"
  inp = raw_input()
  cond.acquire()
  to_eval.extend(string.split(inp, ";"))
  cond.notifyAll()
  cond.release()
```

Результат выполнения:

```
главный вводите:
1 ждет:
2 ждет:
500;400;450;600
главный вводите:
1 активен:
1 выч: len(500!) = 1135
2 активен:
1 выч: len(450!) = 1001
2 выч: len(400!) = 869
1 ждет:
2 выч: len(600!) = 1409
2 ждет:
500;400;450;600
главный вводите:
1 ждет:
2 ждет:
1 выч: len(500!) = 1135
1 выч: len(450!) = 1001
2 выч: len(400!) = 869
1 ждет:
2 выч: len(600!) = 1409
2 ждет:
```

Если вместо числа ввести что-то другое, один из потоков вылетит из-за возникшего исключения и уже больше не выдаст "`ждет:`". Если сбой случился в обоих потоках, главный поток будет продолжать вводить значения, которые уже некому обработать.

9.7.14. Модуль *thread*

Модуль `thread` предоставляет низкоуровневый доступ к потокам управления. Сразу следует оговориться, что не всякая система поддерживает потоки управления. Их поддерживают системы Windows, SGI IRIX, Solaris 2, Linux и другие UNIX-системы, имеющие POSIX-реализацию потоков управления (pthreads). В модуле `thread` доступны следующие функции.

- `thread.start_new(function, args[, kwargs])`

  Создание нового потока для исполнения функции `function` с кортежем аргументов `args` и словарем именованных аргументов `kwargs`. Поток завершается по возврату из функции. В случае возникновения исключения, которое не перехвачено внутри функции, на стандартный вывод ошибок печатается трассировка стека, после чего поток завершается (другие потоки при этом продолжают работать). При поступлении сигнала прерывания от клавиатуры исключение `KeyboardInterrupt` может поймать любой поток, если модуль `signal` не был импортирован, в случае чего это исключение поступает только главному потоку. Имеет синоним `start_new_thread()`.

- `thread.exit()`

  Возбуждает исключение `SystemExit`, которое, если оказалось неперехваченным, завершает исполняющийся поток. Раньше эта функция называлась `exit_thread()`. Кстати, вызов `sys.exit()` или исключения `SystemExit` эквивалентны вызову `thread.exit()`.

- `get_ident() -> id`

  Возвращает идентификатор потока, в котором исполняется эта функция. Идентификатор уникален в любой данный момент времени.

- `thread.allocate_lock() -> lockobj`

  Для своего взаимодействия потоки используют объекты-замки́. Замки́ устроены так, что в любой момент времени данным замко́м может обладать не более чем один поток. В табл. 9.54 приведены методы объекта-замка. Модуль предоставляет константу `LockType` для обозначения типа объекта-замка. Разумеется, программист должен позаботиться о расстановке вызовов методов `acquare()` и `release()` при входе и выходе во фрагменты кода, которые не должны выполняться двумя или более потоками одновременно.

Таблица 9.54. *Методы объекта-замка*

Метод	Описание
`acquire([blocking]) -> 0, 1`	Запрашивает замок. При необходимости ждет снятия замка другим потоком. Возвращает `None`. Если задан аргумент `blocking`, и он равен нулю, ожидания не происходит, а метод сразу возвращается со значением `0` (замо́к не занят) или `1` (замо́к занят). В случае ненулевого аргумента `blocking` действия аналогичны вызову `acquire()` без параметра, но возвращает `1`, а не `None`

Таблица 9.54 *(окончание)*

Метод	Описание
`release()`	Снимает запрошенный ранее (этим или другим потоком) замóк, или возбуждает исключение `thread.error`, если делается попытка снять незанятый замóк.
`locked() -> 0, 1`	Возвращает статус замкá: `0` — свободен, `1` — занят

Документация по Python предупреждает, что использование потоков имеет определенные особенности.

- Не все встроенные функции, блокированные ожиданием ввода, позволяют другим потокам работать, хотя функции `time.sleep()`, `select.select()`, метод `read()` файловых объектов позволяют.
- Невозможно прервать метод `acquire()`, т. к. исключение `KeyboardInterrupt` возбуждается только после возврата из этого метода.
- Очень нежелательно, чтобы главный поток завершался раньше других потоков, т. к. в этом случае не будут выполнены необходимые деструкторы и даже части `finally`. Это связано с тем, что почти все системы завершают приложение, у которого завершился главный поток.

Пример в листинге 9.38 достаточно традиционен для иллюстрации параллельного выполнения потоков. Несколько потоков наперегонки увеличивают размеры своего прямоугольника.

Листинг 9.38. Файл race.py. Гонки между потоками управления

```
import thread, time, sys
from Tkinter import *

# задаем дистанцию и цвета бегунов
distance = 300
colors = ["Red","Orange","Yellow","Green","Blue","DarkBlue","Violet"]
nrunners = len(colors)

def quit():
  """Выход из программы"""
  tk.quit()
  sys.exit(0)

# окно и основа для прямоугольников
tk = Tk()
```

```
tk.title("Гонки")
c = Canvas(tk, width=distance, height=len(colors)*20, bg="White")
c.pack()

# прямоугольники
rects = []
for i in range(nrunners):
  rects.append(c.create_rectangle(0, i*20, 0, i*20+10, fill=colors[i]))

b = Button(text="Go", command=tk.quit)  # кнопка пуска
b.pack(side=LEFT)
qb = Button(text="Quit", command=quit)  # кнопка выхода
qb.pack(side=RIGHT)

def run(n):
  """Программа бега n-го участника"""
  while 1:
    graph_lock.acquire()
    positions[n] = positions[n] + 1  # двигаемся на шаг
    for i in range(1000):
      pass
    if positions[n] == distance:  # если добежал до финиша
      if not champion:   # и чемпионское место пусто
        champion = colors[n]
      graph_lock.release()
      break
    graph_lock.release()

def prepare():
  """На старт!"""
  champion = None
  for i in range(nrunners):
    thread.start_new_thread(run, (i,))

graph_lock = thread.allocate_lock()
while 1:
  b.config(state=NORMAL); qb.config(state=NORMAL)
  tk.mainloop()  # ждем действий зрителя
  b.config(state=DISABLED); qb.config(state=DISABLED)
```

```
  graph_lock.acquire()
  positions = [0]*nrunners  # На старт!
  prepare()                 # Внимание!
  graph_lock.release()      # Марш!
  # главный поток (судья) ждет, пока финишируют все участники
  while reduce(lambda x, y: x+y, positions, 0) < distance*nrunners:
    graph_lock.acquire()
    for n in range(nrunners):
      c.coords(rects[n], 0, n*20, positions[n], n*20+10)
    tk.update_idletasks()
    graph_lock.release()
  for n in range(nrunners):
    c.coords(rects[n], 0, n*20, positions[n], n*20+10)
  b.config(bg=champion)  # окрашиваем кнопку цвет победителя
  tk.update_idletasks()
```

9.8. Простейшие базы данных. Архиваторы

В этом разделе рассматривается работа с простейшими базами данных — хэшами — которые умеют эффективно получать значение по заданному ключу. Кроме того, рассматриваются модули для доступа к архивам и сжатым данным.

9.8.1. Модуль *anydbm*

Модуль `anydbm` предоставляет интерфейс к хэшам — простейшим базам данных, работающим по схеме ключ-значение. Таковыми являются: dbm, gdbm, ndbm и др. Пример в листинге 9.39 (очень похожий на аналогичный пример в *разд. 9.2.10* в листинге 9.15) покажет предоставляемый модулем интерфейс:

Листинг 9.39. Файл database.py. Демонстрация работы с простейшей БД

```
import anydbm, os
filename = "mydat.dat"  # -- имя файла для хранения базы данных

if not os.path.exists(filename):  # если необходимо,
  anydbm.open(filename, "c")      # создание базы данных
```

```
data = "[1,2,3,4]"      # -- данные (обязательно строка)
key = "dat"             # -- ключ (строка)
d = anydbm.open(filename, "w")    # открытие базы данных
d[key] = data           # сохранить данные под ключом key
                        # (затирает старое значение, если оно было)
data = d[key]           # загрузить значение по ключу
len(d)                  # получить количество записей в базе
d.sync()                # запись изменений на диск
del d[key]              # удалить ключ и значение
flag = d.has_key(key)   # проверка наличия ключа
list = d.keys()         # список ключей
d.close()               # завершение работы с базой данных
```

При открытии базы данных `anydbm` автоматически определяет ее тип (с помощью описанного ниже модуля `whichdb`) и использует соответствующий модуль (`dbhash`, `gdbm`, `dbm` или `dumbdbm`). При создании новой базы данных используется первый существующий модуль из приведенного списка.

9.8.2. Модуль *whichdb*

Модуль `whichdb` определяет к какому типу относится хэш (база данных).

```
whichdb.whichdb(file) -> s
```

Возвращает `"dbhash"`, `"gdbm"`, `"dbm"` или `"dumbdbm"`, если файл file является базой данных одного из этих типов, `""`, если тип не может быть определен и `None`, если файл не может быть прочитан (например, не существует).

Примечание

Определение типа базы данных не означает, что нужный модуль установлен или что файл будет читаться с помощью этого модуля.

9.8.3. Модуль *gzip*

Модуль `gzip` обеспечивает последовательный доступ к файлам, сжатым программой `gzip`.

Объект класса `gzip.GzipFile` является файлоподобным объектом *(см. разд. 8.2.9)*. В отличие от обычного файла, `gzip`-объект не имеет методов `tell()`, `seek()` для произвольного доступа к файлу.

Синтаксис конструктора следующий.

```
GzipFile([имя[, режим[, сжатие[, файл_объект]]]]) -> gzip-объект
```

Конструктор gzip-объектов позволяет работать не только с файлами на диске, но и любыми файлоподобными объектами (задается параметром файл_объект). Если файл_объект не задан или None, открывается файл с именем имя. Если файл_объект задан, имя используется для соответствующего поля в заголовке gzip-файла. Уровень сжатия задается параметром сжатие и может варьироваться от самого быстрого, но слабого (1) до самого медленного, но максимального — 9. Параметр режим может быть одним из: "r", "rb", "a", "ab", "w" или "wb", аналогично режиму для открытия обычных файлов (лучше использовать версии с "b"): "r" — чтение, "a" — добавление, "w" — запись.

При закрытии gzip-объекта методом close() файл файл_объект не закрывается. Это позволяет дописывать в него дополнительную информацию после сжатых данных. Таким образом, иногда о закрытии этого объекта нужно позаботиться дополнительно.

Модуль также предоставляет функцию для открытия сжатого файла.

```
gzip.open(имя[, режим[, сжатие]]) -> gzip-объект
```

По умолчанию режим равен "rb", а сжатие — 9.

В следующем примере мы открываем сжатый файл и записываем его содержимое в другой файл, за исключением пустых строк (строк из пробельных символов):

Листинг 9.40. Пример обработки сжатых файлов

```
import gzip, string
f1 = gzip.open("file.gz")            # из этого файла читаем
f2 = gzip.open("file_res.gz", "wb")  # в этот пишем
# Далее можно забыть, что f1 и f2 архивы, считая
# их файлами последовательного доступа
while 1:
  line = f1.readline()
  if not line:
    break
  if string.strip(line):  # во второй файл пишем только
    f2.write(line)        # непустые строки
f1.close()
f2.close()
```

9.8.4. Модуль *zlib*

Модуль `zlib` предоставляет методы *сжатия (компрессии) данных*, совместимые с `gzip`, посредством использования библиотеки `zlib`. В модуле определены следующие функции.

- `zlib.adler32(s[, prevsum]) -> adlersum`

  Вычисляет контрольную сумму строки `s` по алгоритму Adler-32. Этот вид суммы почти так же надежен, как и CRC-32 *(см.* ниже), но может быть вычислен значительно быстрее. Указание `prevsum` (предыдущей контрольной суммы) позволяет вычислять контрольную сумму данных по частям.

- `zlib.compress(s[, level]) -> sc`

  Возвращает результат сжатия строки `s` (уровень сжатия `level` от `1` (бóльшая скорость) до `9` (бóльшее сжатие), по умолчанию `6`).

- `zlib.compressobj([level]) -> zobj`

  Возвращает объект-компрессор для сжатия большого объема данных по частям. Параметр `level` как и для `zlib.compress()`.

- `zlib.crc32(s[, prevsum]) -> crcsum`

  Вычисляет контрольную сумму CRC32 (Cyclic Redundancy Check, циклический избыточный контроль) строки `s`. Указание `prevsum` позволяет вычислять контрольную сумму данных по частям.

- `zlib.decompress(sc[, wbits[, bufsize]]) -> s`

  Возвращает исходную строку `s` из сжатых данных `sc`. Параметр `wbits` может быть в пределах от `8` до `15` (по умолчанию 15). Параметр `bufsize` определяет начальный размер буфера для декомпрессированных данных, по умолчанию `16384`.

- `zlib.decompressobj([wbits]) -> dobj`

  Возвращает объект-декомпрессор для обработки больших объемов сжатых данных. Параметр `wbits` имеет то же значение, что и у функции `zlib.decompress()`

Методы объекта-компрессора приведены в табл. 9.55, а объекта-декомпрессора — в табл. 9.56.

Таблица 9.55. *Методы объекта-компрессора*

Метод	Описание
`compress(s) -> sc`	Сжимает данные строки, возвращая часть сжатых данных. Результаты последовательных вызовов этого метода (и метода `flush()`) дают полный массив сжатых данных

Таблица 9.55 *(окончание)*

Метод	Описание
`flush([mode]) -> sc`	Вызывает досжатие данных объекта, возвращая все сжатые данные без остатка. Режим `mode` может быть одним из следующих: `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` и `Z_FINISH`. Первые два режима позволяют продолжить сжатие данных объектом-компрессором, а `Z_FINISH` говорит о завершении сжатия, после чего метод `compress()` больше не может использоваться

Таблица 9.56. *Метод объекта-декомпрессора*

Метод	Описание
`decompress(sc) -> s`	Возвращает часть сжатых данных из содержавшихся в `sc`. Объединенные результаты последовательных вызовов этого метода и метода `flush()` дают исходные данные, какими они были до сжатия
`flush() -> s`	Возвращает остатки декомпрессированных данных. После вызова этого метода объект-декомпрессор больше не может быть использован

Объект-декомпрессор также имеет атрибут `unused_data`, в котором оказываются лишние данные, поданные на вход. Если размер декомпрессируемых данных неизвестен, появление непустой строки в `unused_data` будет означать окончание сжатых данных.

Пример в листинге 9.41 демонстрирует `zlib` в действии.

Листинг 9.41. Файл zipme.py. Компрессия и декомпрессия данных

```
import zlib
# сжимаем большую, но тривиальную строку
s = "A"*1000
print len(s), zlib.crc32(s)
s1 = zlib.compress(s)
print len(s1)
# получаем исходную строку
s0 = zlib.decompress(s1)
print len(s0), zlib.crc32(s0)
```

Результат выполнения:

```
1000 1369452033
17
1000 1369452033
```

Исходную строку удалось сжать более чем в 50 раз. Помимо этого, можно увидеть, что CRC32-суммы совпадают, а значит данные восстановлены из сжатой строки правильно.

9.8.5. Модуль *zipfile*

ZIP-архивы давно стали самым популярным форматом для архивирования и сжатия файлов, предназначенных для открытия на различных компьютерных платформах. Модуль `zipfile` предоставляет возможности:

- чтения и записи ZIP-файлов, класс `ZipFile`;
- создания архивов библиотек на Python, класс `PyZipFile`;
- предоставления информации о файлах в ZIP-архиве, класс `ZipInfo`.

Конструктор класса `ZipFile` вызывается следующим образом:

```
zipfile.ZipFile(filename[, mode[, compression]]) -> zf
```

Открывает ZIP-файл `filename` в заданном `mode` режиме: `"r"` — для чтения (по умолчанию), `"w"` — для записи (данные существующего файла при этом пропадают), `"a"` — для добавления. Параметр `compression` может быть `zipfile.ZIP_STORED` (по умолчанию) или `zipfile.ZIP_DEFLATED` и указывает уровень сжатия.

В табл. 9.57 приведены методы, присущие экземплярам класса `ZipFile`.

Таблица 9.57. *Методы объектов класса* `ZipFile`

Метод	Описание
`close()()`	Закрывает файл с архивом. Должен быть обязательно вызван до выхода из программы
`getinfo(name) -> zi`	Возвращает `ZipInfo`-объект `zi` для указанного имени `name`, хранящегося в архиве
`infolist() -> zi_lst`	Возвращает список `zi_lst` из `ZipInfo`-объектов в данном архиве
`namelist() -> nm_lst`	Возвращает список `nm_lst` имен файлов, хранящихся в архиве
`printdir()`	Печатает содержание архива на `sys.stdout`
`read(name) -> contents`	Возвращает содержимое файла `name` в виде строки байтов `contents`. Для успеха этой операции архив должен был быть открыт в режиме чтения или добавления

Таблица 9.57 *(окончание)*

Метод	Описание
`testzip() -> res`	Проверяет целостность архива и возвращает имя первого поврежденного файла или `None`, если архив цел. Для проверки целостности используются CRC
`write(filename [, arcname [, compression]])`	Записывает файл filename в архив. Если задано `arcname`, то в архиве файл будет известен под этим именем. Параметр `compression` задает уровень сжатия и может быть равен `zipfile.ZIP_STORED` или `zipfile.ZIP_DEFLATED`. По умолчанию используется уровень сжатия, заданный в конструкторе `Zipfile`-объекта. Для успеха этой операции архив должен быть открыт в режиме записи или добавления
`writestr(zinfo, bytestr)`	Записывает строку байтов `bytestr` в архив. Информация об этом "файле" передается в виде `ZipInfo`-объекта и должна содержать по крайней мере имя файла, дату и время

В следующем примере (листинг 9.42) мы создаем файл arc.zip, записывая в него несколько файлов.

Листинг 9.42. Файл ziparc.py. Пример работы с ZIP-файлом

```
import zipfile

zf = zipfile.ZipFile("arc.zip", "w", zipfile.ZIP_DEFLATED)
zf.write("mydata.dat", "data/mydata.dat")
zf.write("ziparc.py", "bin/ziparc.py", zipfile.ZIP_STORED)
zf.close()

zf = zipfile.ZipFile("arc.zip", "r")
zf.printdir()
print zf.namelist()
print len(zf.read("bin/ziparc.py"))
zf.close()
```

Результат выполнения:

```
File Name                                             Modified             Size
data/mydata.dat                                2001-08-06 12:18:20         5054
bin/ziparc.py                                  2001-08-06 12:20:26          306
['data/mydata.dat', 'bin/ziparc.py']
306
```

Замечание

Если в режиме добавления снова записать существующий файл в архив, в архиве будет две версии одного и того же файла.

Конструктор класса `PyZipFile` принимает те же аргументы, что и `ZipFile`, но добавляет следующий специфичный метод

```
writepy(pathname[, basename])
```

Ищет .py-файлы на указанном пути `pathname` и добавляет в архив соответствующие .pyc- и {pyo}-файлы. Этот метод понимает, как обращаться с пакетами и подпакетами.

Наконец, рассмотрим атрибуты экземпляров класса `ZipInfo`, они приведены в табл. 9.58. (Напомним, что эти объекты создаются и используются объектами класса `ZipFile` и соответствуют информации об одном файле ZIP-архива.)

Таблица 9.58. *Атрибуты объектов класса* `ZipInfo`

Атрибут	Описание
`filename`	Имя файла в архиве
`date_time`	Дата и время последней модификации файла. Кортеж из шести значений, соответствующих первым шести элементам кортежа времени *(см. разд. 9.5.1)*: год, месяц, день, час, минута, секунда
`compress_type`	Тип компрессии
`comment`	Комментарий
`extra`	Дополнительные данные
`create_system`	Система, создавшая архив
`create_version`	Версия PKZIP, создавшая архив
`extract_version`	Версия PKZIP, необходимая для извлечения файла из архива
`reserved`	Зарезервированное поле, должно содержать ноль
`flag_bits`	Биты флагов
`volume`	Номер тома с заголовком файла
`internal_attr`	Внутренние атрибуты
`external_attr`	Внешние атрибуты файла
`header_offset`	Смещение в байтах файлового заголовка
`file_offset`	Смещение в байтах данных файла
`CRC`	CRC-32 исходного (несжатого) файла

Таблица 9.58 (окончание)

Атрибут	Описание
compress_size	Размер сжатых данных
file_size	Размер несжатых данных

Детальное объяснение этих полей может быть найдено на сайте **http://www.pkware.com**.

9.9. Модули для UNIX

Python имеет целый набор модулей, специфичных для UNIX-систем (и систем, поддерживающих стандарт POSIX).

9.9.1. Модули *pwd* и *grp*

В большинстве UNIX-систем есть специальный файл (обычно /etc/passwd), в котором хранится информация о пользователях. (Раньше в этом файле хранились и зашифрованные пароли, но теперь для этого чаще используется другой файл). В модуле `pwd` собраны функции для работы с информацией о пользователях (табл. 9.59).

Таблица 9.59. *Функции модуля* `pwd`

Вызов функции	Описание действия
pwd.getpwnam(string)	Выводит кортеж информации о пользователе по его имени
pwd.getpwuid(int)	То же по идентификатору пользователя
pwd.getpwall()	Список кортежей с информацией о всех пользователях в системе

Кортеж информации о пользователе имеет следующие поля:

`(логин, пароль, uid, gid, gecos, домашний каталог, оболочка)`

где `uid` — идентификатор пользователя; `gid` — идентификатор основной группы пользователя; `gecos` — может содержать имя пользователя, адрес, телефон и прочую контактную информацию.

```
>>> from pwd import *
>>> print getpwnam('root')
('root', 'x', 0, 0, 'root', '/root', '/bin/bash')
```

```
>>> print getpwnam('root')
('root', 'x', 0, 0, 'root', '/root', '/bin/bash')
```

Пользователи могут быть отнесены к одной или нескольким группам, что записывается в файле /etc/group. Модуль `grp` дает доступ к данным в файле. В табл. 9.60 описаны функции, предоставляемые модулем `grp`.

Таблица 9.60. *Функции модуля* `grp`

Вызов функции	Описание действия
`grp.getgrnam(string)`	Выводит кортеж информации о группе по ее имени
`grp.getgrgid(int)`	То же по числовому идентификатору группы
`grp.getgrall()`	Список кортежей с информацией о всех группах, определенных в системе

Кортеж информации о группе имеет следующие поля:

```
(имя, пароль, gid, список пользователей группы)
```

где `gid` — идентификатор группы пользователя.

9.9.2. Модуль *fcntl*

Модуль `fcntl` предоставляет операции управления файловыми дескрипторами и файловым вводом/выводом, являясь интерфейсом к системным вызовам `fcntl()` и `ioctl()` в UNIX. Напомним, что файловый дескриптор открытого файла или сокета можно получить с помощью метода `fileno()`. Отдельные модули `FCNTL` и `IOCTL` содержат константы, необходимые для работы функций модуля `fcntl`.

- `fcntl.fcntl(fd, op, [arg]) -> result`

  Выполняет заданную операцию `op` над файловым дескриптором `fd`. Аргумент `arg` (целое число или строка длиной до 1024 байтов) является необязательным и по умолчанию равен `0`. Если `arg` — строка, результат возвращается в строке такой же длины.

- `fcntl.flock(fd, op)`

  Выполняет операцию `op` (табл. 9.61) *захвата файлового дескриптора (file lock)* `fd`.

  Следующий пример использует `fcntl.flock()` для запирания файла на период записи:

  ```
  import fcntl
  f = open("file.dat", "w")
  fcntl.flock(f.fileno(), fcntl.LOCK_EX)
  ```

```
f.write("data")
fcntl.flock(f.fileno(), fcntl.LOCK_UN)
f.close()
```

- `fcntl.lockf(fd, op, len=0, start=0, whence=0)`

  Аналогична `flock()`, но позволяет захватывать конкретные места в файле. Параметр `len` — количество захватываемых байтов (по умолчанию захват действует до конца файла). Параметр `start` задает начало захватываемых байтов относительно точки отсчета, задаваемой параметром `whence`, который может принимать следующие значения: `1` (от текущей позиции указателя), `2` (от конца файла) или `0` (от начала файла).

Таблица 9.61. *Константы, обозначающие операции в функции* `flock()`

Константа	Описание операции
`fcntl.LOCK_SH`	Совместный захват. Более чем один процесс может захватить данный файл в одно и то же время
`fcntl.LOCK_EX`	Монопольный захват. Только один процесс может использовать такой захват для конкретного файла в одно и то же время
`fcntl.LOCK_UN`	Снятие захвата
`fcntl.LOCK_NB`	Модификатор, исключающий блокировку процесса при применении захвата (non-blocking lock). Может быть добавлен побитовым ИЛИ к любой из констант, описанных выше. Если не задан, процесс блокируется и ждет освобождения файла

- `fcntl.ioctl(fd, op, [arg]) -> result`

  Выполняет операцию `op`, управляющую вводом/выводом файла с дескриптором `fd`. Эта функция обычно применяется для управления файлами-устройствами. Параметр `arg` аналогичен тому, что описан выше для функции `fcntl()`: целое число или строка длиной до 1024 байтов.

Пример в листинге 9.43 иллюстрирует применение функции `lockf()` для блокирования конкретного байта в файле.

Листинг 9.43. Файл setlock.py. Применение функции `lockf()`

```
#!/usr/bin/env python
import fcntl, os, sys, time
f = open("set.dat", "r+")  # открываем файл для чтения и записи
fd = f.fileno()  # определяем его файловый дескриптор
flen = os.stat("set.dat")[6]  # длина файла
```

```
# режим можно будет задавать из командной строки
if len(sys.argv) > 1 and sys.argv[1] == "-x":
  exclusive = fcntl.LOCK_NB
else:
  exclusive = 0

def toggle_byte(n):
  """Переключает байт с "0" на "1" и обратно"""
  # ставим замок на n-й байт в файле
  fcntl.lockf(fd, fcntl.LOCK_EX | exclusive, 1, n, 0)
  try:
    # читаем значение байта
    f.seek(n, 0)
    v = f.read(1)
    # меняем значение байта с "0" на "1" и наоборот
    v = (v == "1") and "0" or "1"
    # записываем новое значение на старое место в файле
    f.seek(n, 0)
    f.write(v)
  finally:
    # снимаем замок, что бы ни случилось
    fcntl.lockf(fd, fcntl.LOCK_UN, 1, n, 0)

for i in xrange(1000):
  for l in range(flen):
    try:
      toggle_byte(l)
    except KeyboardInterrupt:
      sys.exit("Exiting")
    except IOError:
      # Ресурс занят. Пропускаем.
      sys.stderr.write("!")
      time.sleep(0.01)
```

Чтобы увидеть работу замков и блокировок, нужно запустить два или больше экземпляров программы. Не забудьте создать файл set.dat. При запуске программы `setlock.py` с ключом `-x` вы увидите спорадически высыпающие на экран восклицательные знаки, сигнализирующие о том, что программа пыталась обратиться к байту, на который другой программой был наложен замок.

Без ключа `-x` программа не выдает никаких восклицаний, а ждет освобождения ресурса. Чем больше запущено процессов, тем больше вероятность конфликтов и тем больше восклицательных знаков выводится на консоль.

Следует заметить, что поддержка замков — дело сугубо добровольное. Это означает, что процесс, не использующий этого механизма, может запросто получить доступ к файлу в любом режиме, без всяких блокировок.

Еще один скользкий момент касается использования функции `os.fork()`, т. к. в некоторых системах монопольные замки наследуются как родительским, так и дочерним процессом.

9.9.3. Модуль *resource*

Модуль `resource` предоставляет возможность контроля системных ресурсов в UNIX.

Следующие функции получают и устанавливают действующие ограничения по ресурсам.

```
resource.getrlimit(resource) -> (softlimit, hardlimit)
resource.setrlimit(resource, (softlimit, hardlimit))
```

где `resource` — тип ресурса, см. константы в табл. 9.62 (не все указанные типы ресурсов поддерживаются всеми UNIX-системами); `softlimit` — мягкий предел; `hardlimit` — жесткий предел.

В случае, когда `resource` указан неверно, мягкий предел превышает жесткий или приложение пытается повысить свой жесткий предел, не имея на это привилегий (действующий UID пользователя root), возбуждается исключение `ValueError`. Возможны и другие исключения. Для указания максимально возможного значения можно использовать `-1`.

В следующем примере поставлен очень низкий предел количества открытых файлов (4), поэтому оказывается возможным открыть только один файл (в дополнение к `sys.stdin`, `sys.stdout` и `sys.stderr`, которые открыты по умолчанию).

```
>>> import resource
>>> resource.setrlimit(resource.RLIMIT_OFILE, (4, 5))
>>> f1 = open("/dev/null")
>>> f2 = open("/dev/zero")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 24] Too many open files: '/dev/zero'
>>> # слишком много открытых файлов
```

Таблица 9.62. *Константы для описания типа ресурса*

Константа	Значение
`RLIMIT_CORE`	Максимальный размер в байтах файла дампа памяти `core`
`RLIMIT_CPU`	Максимальное количество процессорного времени, которое данный процесс может использовать. В случае превышения этого предела процессу посылается сигнал `SIGXCPU` *(см. разд. 9.7.10)*
`RLIMIT_FSIZE`	Максимальный размер файла, который может обрабатывать данный процесс
`RLIMIT_DATA`	Максимальный размер области данных процесса
`RLIMIT_STACK`	Максимальный размер стека. В многопоточном приложении относится к главному потоку
`RLIMIT_RSS`	Максимальный размер физической памяти процесса (RSS, resident set size)
`RLIMIT_NPROC`	Максимальное количество процессов, которые может породить текущий процесс
`RLIMIT_NOFILE`, `RLIMIT_OFILE`	Максимальное количество открытых файлов
`RLIMIT_MEMLOC`	Максимальное адресное пространство, которое может быть зафиксировано в памяти
`RLIMIT_VMEM`	Максимальный размер виртуальной памяти, доступной процессу

Следующая функция получает текущую информацию о ресурсах процесса и/или дочерних процессов:

```
resource.getrusage(who) -> tuple
```

Параметр `who` указывает, чьи ресурсы будут считаться и может принимать значения `resource.RUSAGE_SELF` (процесс), `resource.RUSAGE_CHILDREN` (дочерние процессы) или `resource.RUSAGE_BOTH` (процесс и его дочерние процессы). Структура возвращаемого функцией кортежа описана в табл. 9.63.

Таблица 9.63. *Описание ресурсов из кортежа, возвращаемого функцией* `resource.getrusage()`

Индекс	Описание
0	Время работы процесса
1	Время работы ОС (по обслуживанию этого процесса)
2	Максимальная физическая память процесса (RSS)
3	Размер разделяемой области памяти

Таблица 9.63 (окончание)

Индекс	Описание
4	Размер неразделяемой памяти
5	Размер стека
6	Обращение к отсутствующим в памяти страницам, не требующим ввода/вывода
7	То же требующим ввода/вывода
8	Количество обращений к свопу
9	Блочные операции ввода
10	Блочные операции вывода
11	Отправлено сообщений
12	Получено сообщений
13	Получено сигналов
14	Количество произвольных переключений константы
15	Количество непроизвольных переключений константы

```
resource.getpagesize() -> pagesize
```

Функция возвращает размер страницы (в байтах), который использует операционная система.

9.9.4. Модуль *termios*

Для управления терминалами (tty-устройствами) по стандарту POSIX в UNIX используется модуль `termios`. Терминальные устройства устроены сложнее файлов и потому требуют некоторых дополнительных управляющих воздействий. Напомним, что узнать, является ли файловый или файлоподобный объект tty-устройством можно с помощью метода `isatty()` файлового объекта.

Примечание

В старых версиях Python константы определены в модуле `TERMIOS`, а не `termios`.

Модуль `termios` имеет следующие функции.

- `termios.tcdrain(fd)`

  Ожидает передачи всех выведенных в файловый дескриптор `fd` данных.

- `termios.tcflow(fd, action)`

  Приостанавливает или возобновляет ввод и вывод в файловый дескриптор `fd`. Параметр `action` может быть одним из `termios.TCOOFF` (остановить вывод), `termios.TCOON` (возобновить вывод), `termios.TCIOFF` (остановить ввод) или `termios.TCION` (возобновить ввод).

- `termios.tcflush(fd, ioqueue)`

  Отбрасывает данные в очереди ввода `termios.TCIFLUSH`, вывода `termios.TCOFLUSH` или обоих очередях `termios.TCIOFLUSH` файлового дескриптора `fd`.

- `tcsendbreak(fd, duration)`

  Посылает прерывание (break) продолжительностью `duration`, если tty-устройство использует асинхронный ввод/вывод, в противном случае сразу возвращается. Если `duration` задана равной нулю, продолжительность прерванного состояния составляет 0,25—0,5 секунд. В остальных случаях это состояние длится `duration*C` секунд, где `C` зависит от платформы. На Linux `C` находится в диапазоне от `0.25` до `0.5`.

- `termios.tcgetattr(fd) -> attr_list`

  Функция возвращает список атрибутов `attr_list` файлового дескриптора `fd` следующего формата.

  ```
  [iflag, oflag, cflag, lflag, ispeed, ospeed, cc_list]
  ```

  где `iflag` — режимы ввода; `oflag` — режимы вывода; `cflag` — режимы управления; `lflag` — местные режимы; `ispeed` — скорость ввода; `ospeed` — скорость вывода; `cc_list` — список специальных символов.

Для задания режимов в модуле определены многочисленные константы. Значения этих констант можно посмотреть в документации к ОС UNIX по команде `man termios 3`. Примеры: `termios.IGNBRK` — игнорировать прерывание ввода (используется в `iflag`), `termios.ISTRIP` — срезать восьмой бит при вводе (там же), `termios.ONLCR` — преобразовывать LF в CR-LF на выводе (`oflag`), `termios.ECHO` — отображать вводимые символы (`lflag`). Наконец, для установления скорости ввода/вывода предусмотрены `B`-константы `B0`, ..., `B19200`, `B38400`, `B57600`, `B115200` и выше. (Установление скорости `B0` приводит к завершению соединения.)

Управление терминалом происходит с помощью специальных символов. Список специальных символов содержит односимвольные строки, возможно, за исключением элементов в позициях `termios.VTIME` и `termios.VMIN`. Индексами к этому списку служат определенные в `termios` константы `VINTR`, `VQUIT`, `VERASE`, `VKILL`, `VEOF`, `VTIME`, `VMIN`, `VSWTC`, `VSTART`, `VSTOP`, `VSUSP`, `VEOL`, `VREPRINT`, `VDISCARD`, `VWERASE`, `VLNEXT`, `VEOL2`.

Следующая функция позволяет задавать свойства терминала `fd`.

```
tcsetattr(fd, when, attr_list)
```

Параметр `when` указывает, когда новые параметры должны вступить в силу: `termios.TCSANOW` (немедленно), `termios.TCSADRAIN` (после того как будет передано все из очереди вывода) или `termios.TCSAFLUSH` (после передачи всего из очереди вывода и отбрасывания очереди ввода). Параметр `attr_list` имеет такой же формат, что и возвращаемый `termios.tcgetattr()`. Пример в листинге 9.44 показывает как можно заменить символ "забой" на другой. Запустите программу в UNIX и попробуйте при вводе строки использовать `"-"` как клавишу "забоя".

Листинг 9.44. Файл myBS.py. Замена клавиши "забоя" на терминале

```
import sys, termios
stdin_fd = sys.stdin.fileno()
if sys.stdin.isatty():  # если имеем дело с tty
  cc = termios.tcgetattr(stdin_fd)
  old_erase = cc[-1][termios.VERASE]
  print "Old erase:", `old_erase`
  cc[-1][termios.VERASE] = "-"
  print "New erase:", `cc[-1][termios.VERASE]`
  try:
    termios.tcsetattr(stdin_fd, termios.TCSANOW, cc)
    r = sys.stdin.readline()
    print `r`
  finally:  # при любом раскладе, восстанавливаем терминал
    cc[-1][termios.VERASE] = old_erase
    termios.tcsetattr(stdin_fd, termios.TCSANOW, cc)
```

Предупреждение

При работе с настройками терминала бывает очень важно восстанавливать его исходные свойства. Программа, изменяющая свойства терминала, должна запоминать исходные свойства и затем восстанавливать их независимо от возникающих в ней сбоев. Например, в Python можно применять оператор `try-finally` для обязательного восстановления исходных свойств терминала в части `finally` (как это сделано в примере). Восстановить свойства терминала "вслепую" можно с помощью команды `stty sane` системы UNIX.

9.10. Редактирование в командной строке

Библиотека `readline` делает командную строку подчас более удобным интерфейсом, чем графический. Описываемые здесь модули позволяют встраи-

вать возможности readline в программы на Python, а также использовать ее для диалогового режима интерпретатора.

9.10.1. Модуль *readline*

Модуль readline предоставляет привязки к библиотеке GNU readline. В табл. 9.64 описаны доступные из Python функции. *Автозавершением (tab-completion)* называется возможность завершить слово в тексте на основании некоторого списка, что обычно делается нажатием клавиши <Tab>.

Таблица 9.64. *Функции модуля* `readline`

Функция	Описание
`get_begidx() -> i`	Возвращает позицию начала области автозавершения (начало слова, которое пытаются автозавершить)
`get_completer_delims() -> s`	Возвращает строку с разделителями слов, используемыми readline
`get_endidx() -> i`	Возвращает позицию окончания области автозавершения (конец слова)
`get_history_length() -> i`	Возвращает количество строк в истории редактирования
`get_line_buffer() -> s`	Возвращает содержимое буфера редактирования
`insert_text(s)`	Вставляет текст s в командную строку
`parse_and_bind(s)`	Разбирает строку s и исполняет записанную в ней команду как если бы она встретилась в файле инициализации
`read_history_file ([filename])`	Загружает файл с историей редактирования filename. По умолчанию "~/.history"
`read_init_file ([filename])`	Разбирает файл инициализации. По умолчанию используется тот же файл, что был указан ранее
`set_completer([func])`	Устанавливает или убирает функцию автозавершения слов. Функция вызывается как func(text, i) для i от нуля и до тех пор, пока функция возвращает строку. Возвращаемая строка должна являться возможным завершением для text
`set_completer_delims (s)`	Устанавливает символы в s в качестве разделителей слов
`set_history_length (length)`	Устанавливает максимальную длину истории редактирования. Отрицательное значение запрещает усечение истории
`write_history_file ([filename])`	Сохраняет историю редактирования в файле filename. По умолчанию "~/.history"

Пример в листинге 9.45 позволяет организовать автоматическое сохранение истории редактирования между сеансами интерактивной работы с интерпретатором. Для этого приведенный в листинге код необходимо добавить к файлу start.py, *см. разд. 9.10.2*. (Пример работает под UNIX/Linux.)

Листинг 9.45. Файл start1.py. Добавление к файлу start.py

```
import os, atexit
try:
  history = os.path.join(os.environ["HOME"], ".python_history")
  readline.read_history_file(history)
  atexit.register(readline.write_history_file, history)
  del os, history
  print "History loaded."
except:
  print "History NOT loaded."
```

Следует заметить, что перед работой с `readline` полезно ознакомиться с теми возможностями, которые она предоставляет через комбинации "горячих" клавиш. Некоторые из этих клавиш приведены в табл. 9.65.

Таблица 9.65. *"Горячие" клавиши* `readline`

Комбинация клавиш	Обозначение	Действие
<Ctrl>+<B> или <-	`C-b`	Перемещение курсора назад на одно знакоместо
<Ctrl>+<F> или ->	`C-f`	Перемещение курсора вперед на одно знакоместо
<Ctrl>+<D>	`C-d`	Удаление знака под курсором (аналог <Del>)
<Ctrl>+<_> или <Ctrl>+<X> <Ctrl>+<U>	`C-_` или `C-xC-u`	Откатка последней выполненной операции редактирования (откатку можно продолжать вплоть до исходной строки)
<Ctrl>+<A>	`C-a`	Перемещение курсора в начало строки
<Ctrl>+<E>	`C-e`	Перемещение курсора в конец строки
<Alt>+<F>	`M-f`	Перемещение курсора на слово вперед
<Alt>+<B>	`M-b`	Перемещение курсора на слово назад
<Ctrl>+<L>	`C-l`	Очистка экрана (редактируемая строка не изменяется)
<Ctrl>+<R>	`C-r`	Инкрементный поиск назад по истории редактирования (очень полезная команда!)

Примечание

Документация по программному обеспечению GNU использует обозначения во второй колонке таблицы. Буква "C" обозначает <Control> или <Ctrl>, а "M" — <Meta> или <Alt> (как на персональных компьютерах).

Конечно, набор команд для редактирования этим не ограничивается. Интересующиеся могут познакомиться с ним по документации к GNU readline.

9.10.2. Модуль *rlcompleter*

Модуль `rlcompleter` предоставляет возможности автозавершения слов для GNU readline в среде Python. Модуль предоставляет класс `Completer`, который помогает адаптировать `readline` к среде интерпретатора Python для автозавершения идентификаторов.

Чтобы попробовать `rlcompleter` в действии, выполните:

```
>>> import readline, rlcompleter
>>> readline.parse_and_bind("tab: complete")
>>> pri
```

и нажмите клавишу <Tab>. После этого "`pri`" само дополнится до "`print`"!

В табл. 9.66 представлены методы объектов класса `rlcompleter.Completer`.

Таблица 9.66. *Методы объектов класса* `rlcompleter.Completer`

Метод	Описание
`attr_matches (text)`	Вычисляет все возможные продолжения `text`, если `text` содержит точки. Внимание! Эта функция может вызвать код метода `__getattr__` объекта, атрибуты которого считывает функция
`Complete ( text, i)`	Возвращает `i`-е возможное завершение `text` (`i` от нуля) или нестроку. Годится в качестве `func()` для `readline.set_completer` (см. табл. 9.64)
`global_matches (text)`	Возвращает список возможных завершений, если `text` — обычный идентификатор. В список входят зарезервированные слова, имена встроенных функций и имена из `__main__`

Следующий пример показывает, как приспособить `rlcompleter` для постоянного использования в интерактивном режиме Python.

Примечание

Сказанное ниже относится к пользователям UNIX/Linux. Для пользователей Windows имеется библиотека с альтернативным `readline`, которую можно найти в Internet.

Чтобы Python исполнял некоторые команды перед началом работы в диалоговом режиме, можно поместить их в некоторый файл (например, ~/start.py), а затем присвоить полный абсолютный путь к этому файлу переменной окружения PYTHONSTARTUP, например, добавив следующую строку к файлу ~/.bash_profile:

```
export PYTHONSTARTUP=~/start.py
```

Листинг 9.46. Файл start.py. Начальные команды для интерпретатора

```
import readline, rlcompleter
readline.parse_and_bind("tab: complete")
readline.parse_and_bind("set meta-flag On")
readline.parse_and_bind("set convert-meta Off")
readline.parse_and_bind("set output-meta On")
print "Readline loaded."
```

Теперь начало работы с интерпретатором будет выглядеть, например, так:

```
$ python2.1
Python 2.1 (#1, Jun 28 2001, 21:47:26)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.
Readline loaded.
>>>
```

А вы получите возможность использовать автозавершение.

9.11. Отладчик и профайлер

При разработке программы достаточно редко бывает, что она заработает правильно с первого раза. Процесс выявления ошибок в программе называется *отладкой* и может использовать специальное средство — *отладчик*. После того как программа отлажена, вас может не удовлетворить ее быстродействие. Помочь найти узкие места в программе на Python поможет инструмент, называемый *профайлером (profiler)*. Ниже вы найдете краткое описание отладчика и профайлера, доступных в стандартном Python.

9.11.1. Модуль *pdb*

Модуль pdb содержит интерактивный отладчик кода на языке Python. Он имеет следующие возможности:

- построчное исполнение программы;
- установка точек останова (условных и безусловных);

- просмотр стековых фреймов;
- выполнение любого кода на Python в контексте любого стекового фрейма;
- post mortem анализ аварийно завершившейся программы;
- может управляться из программы;
- легко поддается модификации путем наследования от класса `Pdb`.

Ниже перечислены функции модуля `pdb`, которые можно вызывать для отладки в тех или иных случаях.

- `pdb.run(statement[, globals[, locals]])`

  Исполняет оператор `statement` в заданных глобальном и локальном пространствах имен.

- `pdb.runeval(expr[, globals[, locals]]) -> result`

  Вычисляет выражение `expr` и возвращает результат. В остальном аналогична `pdb.run`.

- `runcall(function[, argument, ...]) -> result`

  Вызывает функцию (или метод) `function` с заданными аргументами, возвращает результат.

- `pdb.set_trace()`

  Вызывает интерактивный отладчик в контексте текущего стекового фрейма. Может использоваться как прописанная в коде программы точка останова.

- `pdb.post_mortem(traceback)`

  Начать post mortem отладку на основе данного трассировочного объекта.

- `pdb.pm()`

  Аналогична `pdb.post_mortem` для последнего трассировочного объекта, который берется из `sys.last_traceback`.

Следующий пример иллюстрирует процесс отладки функции с помощью отладчика `pdb`.

Листинг 9.47. Файл myf.py. Файл с отлаживаемой функцией

```
import pdb
def f(x):
  l = []
  for i in range(5):
    l.append(i+x)
  return l
```

Так может выглядеть типичный процесс отладки:

```
>>> pdb.runcall(myf.f, 10)
> /home/rnd/myf.py(1)f()
-> def f(x):
(Pdb) next
> /home/rnd/myf.py(2)f()
-> l = []
(Pdb) list
  1     def f(x):
  2  ->   l = []
  3       for i in range(5):
  4         l.append(i+x)
  5       return l
  6
[EOF]
(Pdb) n
> /home/rnd/myf.py(3)f()
-> for i in range(5):
(Pdb) l
  1     def f(x):
  2       l = []
  3  ->   for i in range(5):
  4         l.append(i+x)
  5       return l
  6
[EOF]
(Pdb) n
...
(Pdb) n
> /home/rnd/myf.py(3)f()
-> for i in range(5):
(Pdb) !l
[10, 11, 12]
```

9.11.2. Модуль *profile*

Использование модуля `profile` позволит измерить, сколько процессорного времени занимает выполнение тех или иных функций в программе. Пример в листинге 9.48 показывает самый простой вариант выполнения профайлинга.

Листинг 9.48. Файл prof_1.py. Простейший пример профайлинга

```
#!/usr/bin/env python
import profile
from math import *

def dist1((x, y), (x1, y1)): return abs(x-x1)+abs(y-y1)
def dist2((x, y), (x1, y1)): return sqrt((x-x1)**2+(y-y1)**2)
def dist3((x, y), (x1, y1)): return sqrt((x-x1)*(x-x1)+(y-y1)*(y-y1))
def dist4((x, y), (x1, y1)): return (x-x1)*(x-x1)+(y-y1)*(y-y1)

def f(N):
  for dist in dist1, dist2, dist3, dist4:
    for i in xrange(N):
      dist((i+1.0, i+2.0), (i+20.0, i+15.0))

profile.run("f(10000)")
profile.run("f(10000)", "profile.dat")
```

Результат выполнения:

```
         40003 function calls in 6.420 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.360    6.360 <string>:1(?)
        1    3.030    3.030    6.360    6.360 prof_eg.py:10(f)
    10000    0.890    0.000    0.890    0.000 prof_eg.py:5(dist1)
    10000    0.850    0.000    0.850    0.000 prof_eg.py:6(dist2)
    10000    0.900    0.000    0.900    0.000 prof_eg.py:7(dist3)
    10000    0.690    0.000    0.690    0.000 prof_eg.py:8(dist4)
        1    0.060    0.060    6.420    6.420 profile:0(f(10000))
        0    0.000             0.000          profile:0(profiler)
```

где `ncalls` — количество вызовов функции; `tottime` — время выполнения кода функции (исключая время, проводимое внутри других функций, вызываемых из данной); `percall` — то же, в пересчете на один вызов; `cumtime` — суммарное время выполнения кода функции; `filename:lineno` — имя файла и номер строки файла с функцией.

Выполнение указанной в кавычках функции "F(10000)" заняло 6,42 секунды процессорного времени. При этом было выполнено 40 003 вызовов функций. По приведенным данным можно увидеть, что наиболее быстрой оказалась функция `dist4()`. Это, скорее всего, связано с тем, что вызов функции в Python обходится дороже выполнения встроенной операции, поэтому две "пустяковые" операции `abs` занимают больше времени, чем выполнение одной более сложной `sqrt`.

Второй вызов `profile.run()` сохраняет статистическую информацию в файле `profile.dat`, откуда ее можно позднее извлечь и проанализировать, что и продемонстрировано в программе `prof_an.py` в листинге 9.49.

Листинг 9.49. Файл prof_an.py. Анализ данных профайлинга

```
import pstats

print "Загружаем стат. данные:"
stat = pstats.Stats('profile.dat')  # получаем объект класса Stat
print "Сортируем по именам и печатаем:"
stat.sort_stats('name')
stat.print_stats()
print "По полному времени выполнения (первые 7):"
stat.sort_stats('cumulative').print_stats(7)
print "Только функции dist*:"
stat.print_stats('dist')
print "Половина списка (в обратном порядке):"
stat.reverse_order()
stat.print_stats(.5)
print "Кто вызывал функции dist:"
stat.print_callers('dist')
print "Что вызывала функция f:"
stat.print_callees('\(f\)')
stat.add("profile.dat", "profile.dat")
stat.print_callees('\(f\)')
```

В табл. 9.67 приведены возможные аргументы для сортировки. Метод `sort_stats()` может принимать несколько таких аргументов. В этом случае сортировка будет производиться сначала по первому ключу, затем среди значений с одинаковыми первыми ключами — по второму, и т. д.

Таблица 9.67. *Аргументы метода* `sort_stats`

Аргумент	Значение
`"calls"`	Количество вызовов
`"cumulative"`	Суммарное время
`"file"`	Имя файла
`"module"`	Имя файла
`"pcalls"`	Количество примитивных вызовов
`"line"`	Номер строки
`"name"`	Имя функции
`"nfl"`	Имя/файл/строка
`"stdname"`	Стандартное имя
`"time"`	Внутреннее время

Примечание

Время и количество вызовов сортируются в порядке убывания, а все остальное — по возрастанию.

Передаваемые методам `print_stats()` (печать статистики), `print_callees()` (печать тех, кого вызывали), `print_callers()` (печать тех, которые вызывали) аргументы ограничивают полный список. Целое число `N` в качестве аргумента означает, что необходимо взять первые `N` записей, число с плавающей запятой от 0 до 1 задает долю записей, которые будут представлены, а строка — задает подстроку, которая должна встретиться в поле `filename:lineno(function)`, правда, скобка требует экранирования с помощью `"\"`. Ограничения действуют в порядке задания аргументов.

Кроме того, объект класса `Stats` позволяет использовать в анализе несколько файлов со статистикой, объединяя результат. Для добавления еще одного файла статистики используется метод `add()`. Он принимает в качестве аргументов имена файлов.

Теперь, когда вам стали известны функции, которые было бы неплохо оптимизировать, вернитесь к *разд. 8.9*, в котором описаны некоторые особенности оптимизации программ на Python.

9.12. Поддержка Internet. Протоколы

Все работающие в сети устройства и программы при взаимодействии друг с другом придерживаются стандартизированных схем передачи данных, называемых *сетевыми протоколами*. Протоколы однозначно определяют допус-

тимые форматы передаваемых данных, особенности их передачи и приема. Сетевые протоколы имеют несколько уровней, каждый из которых строго выполняет свои функции. Уровни используют результаты работы соседних уровней (в виде данных), но практически не вмешиваются в их работу.

Для описания *стека протоколов (protocol stack)* принято использовать семиуровневую модель OSI (Open Systems Interconnection, взаимодействие открытых систем). В табл. 9.68 описаны уровни модели OSI и их отражение в стеках Internet-протоколов TCP/IP, UDP/IP.

Таблица 9.68. *Семиуровневая модель OSI и протоколы Internet*

Уровень	В сети Internet	Описание
Физический уровень	Разные	Поток битов, передаваемых по физической линии. Определяет параметры физической линии
Канальный уровень	Ethernet LLC, PPP, ATM, ...	(Де)кодирует данные в виде потока битов, справляясь с ошибками, возникающими на физическом уровне в пределах физически единой сети
Сетевой уровень	IP	Занимается маршрутизацией информационных пакетов от узла к узлу
Транспортный уровень	TCP, UDP	Обеспечивает прозрачную передачу данных между двумя точками соединения
Сеансовый уровень		Управляет сеансом соединения между участниками сети, начиная, координируя и завершая соединения
Уровень представления		Обеспечивает независимость данных от формы их представления путем соответствующего преобразования форматов. Например, на этом уровне может выполняться прозрачное (с точки зрения вышележащего уровня) (де)шифрование данных
Уровень приложений	HTTP, FTP, telnet, ...	Поддерживает конкретные сетевые приложения для конечного пользователя. На этом уровне все зависит от конкретного типа приложения

В этом разделе мы рассматриваем протоколы уровня приложений и только модуль `socket` посвящен более низким уровням. В сети Internet используется модель `клиент-сервер`, и там, где протоколы несимметричны, рассматриваемые здесь модули относятся к клиентской части. На языке Python можно создавать и соответствующие серверные части. Более того, среди стандартных модулей можно встретить примеры реализации HTTP-сервера (как простого, так и с поддержкой CGI).

9.12.1. Модуль *cgi*

Модуль `cgi` облегчает разработку сценариев *CGI (Common Gateway Interface, общий интерфейс шлюзов)* на языке программирования Python. CGI-сценарии — один из наиболее популярных и, наверное, самый простой способ организации активных Web-сайтов со стороны сервера (server-side). CGI обеспечивает обратную связь от HTTP-клиента, с помощью которой можно передавать различные параметры, результаты заполнения форм и даже файлы. CGI-сценарий запускается HTTP-сервером при обращении к универсальному идентификатору ресурса, который HTTP-сервер считает сценарием. HTTP-сервер передает сценарию запрос клиента (в зависимости от метода, например, на стандартный ввод), сопутствующую информацию в переменных окружения (они доступны из Python в виде словаря `os.environ`), а ожидает специальный заголовок и готовую страницу со стандартного вывода (эта информация отсылается клиенту). Пример в листинге 9.50 показывает простейший CGI-сценарий, который выводит показания часов.

Листинг 9.50. Файл simple.py. Простейший CGI-сценарий

```
#!/usr/bin/python
# узнаем время
from time import strftime, localtime, time
now = strftime("%H:%M (%Z)", localtime(time()))
# выводим заголовочную часть:
print "Content-Type: text/html"
print
# ... и саму HTML-страницу:
print """<HTML><HEAD><TITLE>Результат работы CGI-сценария</TITLE></HEAD>
<BODY><H1>Привет!</H1><P>Сейчас %s</BODY></HTML>""" % now
```

Примечание

Разместите файл simple.py в каталоге, в котором HTTP-сервер хранит сценарии (обычно этот каталог называется `cgi-bin`). Не забудьте исправить первую строку, чтобы она точно отражала местонахождение интерпретатора Python. Сделайте файл `simple.py` читаемым и исполняемым (команда `chmod a+rx simple.py`). Убедитесь, что HTTP-сервер запущен. После этого попробуйте в Web-браузере набрать URL **http://localhost/cgi-bin/simple.py**. Если в конфигурации вашей системы нет никаких осложнений, указанная Web-страница покажет точное время. (Кстати, файл модуля `cgi` — cgi.py — может работать и как CGI-сценарий.) Если у вас возникли проблемы, попробуйте выяснить их по документации к HTTP-серверу.

Модуль `cgi` позволяет программисту абстрагироваться от особенностей передачи параметров через CGI, дав ему возможность работать с полученными данными почти так же просто, как со словарем.

В листинге 9.51 представлен более сложный пример, в котором CGI-сценарий обрабатывает введенные в форму данные (месяц и год) и выводит календарь для заданного месяца.

Листинг 9.51. Файл cal.cgi. Календарь

```
#!/usr/bin/python
import cgi, calendar, string

# выводим заголовочную часть:
print "Content-Type: text/html"
print

form = cgi.FieldStorage()

# если данные формы введены
if form.has_key("month") and form.has_key("year"):
  # проверяем данные, введенные пользователем
  form_ok = 0
  try:
    month = int(form["month"].value)
    year = int(form["year"].value)
    if 1 <= month <= 12 and 1970 <= year <= 2037:
      form_ok = 1
  except:
    pass
  if form_ok:
    # формируем HTML-таблицу с календарем
    month_cal = calendar.monthcalendar(year, month)
    table = ["""<B>%s.%s</B>""" % (month, year),
             """<TABLE BORDER=1>"""]
    for week in month_cal:
      table.append("""\n<TR>""")
      for day in week:
        table.append("""<TD>%s</TD>""" % (day or " "))
      table.append("""</TR>""")
    table.append("""</TABLE>""")
```

```
    message = string.join(table, "")
  else:
    message = """<FONT COLOR="Red">Ошибка!</FONT>"""
else:
  message = ""

# выводим HTML-страницу
print """<HTML><HEAD><TITLE>Календарь</TITLE></HEAD>
<BODY><H1>Календарь</H1><P>%s
<P><FORM ACTION="cal.cgi" METHOD="GET">
Введите:
<br>Месяц: <INPUT SIZE=2 TYPE="text" NAME="month">
<br>Год: <INPUT SIZE=4 TYPE="text" NAME="year">
<br><INPUT TYPE="submit" NAME="b" VALUE="Календарь">
</FORM></BODY></HTML>""" % message
```

На рис. 9.1 показан результат выполнения CGI-сценария `cal.cgi` с параметрами `year=2001` и `month=12`.



Рис. 9.1. Результат работы `cal.cgi` в окне Web-браузера

Класс `cgi.FieldStorage` — основное, что предлагает модуль `cgi`. Вся информация, относящаяся к запросу, доступна через объект этого класса. В одном CGI-сценарии конструктор класса `cgi.FieldStorage` должен вызываться только один раз. Кроме того, если вы используете модуль `cgi`, нежелательно читать стандартный ввод, т. к. это может привести к некорректным данным в `cgi.FieldStorage` когда для передачи данных используется метод `POST`.

В табл. 9.69 приведены атрибуты объектов класса `cgi.FieldStorage`, а в табл. 9.70 — его методы.

***Таблица 9.69.** Атрибуты класса `cgi.FieldStorage`*

Атрибуты	Описание
`disposition`	Расположение содержимого
`disposition_options`	Словарь с опциями, относящимися к расположению содержимого

Таблица 9.69 *(окончание)*

Атрибуты	Описание
`file`	Файловый объект, содержащий данные. Имеет значение `None`, если данные хранятся в строке
`filename`	Имя файла, данное клиентом, или `None`, если не указано
`headers`	Объект, подобный словарю, содержащий все поля заголовочной части запроса
`list`	Список объектов-полей (обычно состоит из объектов класса `MiniFieldStorage`)
`name`	Имя поля, если задано
`type`	Тип содержимого (например, `application/x-www-form-urlencoded`) или `None`, если тип не задан
`type_options`	Словарь опций типа содержимого
`value`	Значение в виде строки. Если содержимое — файл, он перечитывается каждый раз при получении `value`

Объекты класса `MiniFieldStorage` имеют только атрибуты `name` и `value`.

Таблица 9.70. *Методы объектов класса `cgi.FieldStorage`*

Метод	Описание
`keys()`	Возвращает список имеющихся полей
`has_key(key)`	Проверяет наличие ключа `key`
`getvalue(key [, default])`	Выдает значение поля по ключу. В случае, когда имена полей в форме повторяются, возвращается список значений. Если ключ не найден, возвращается значение `default`

Кроме того, `cgi.FieldStorage` поддерживает синтаксис взятия значения по ключу, что было продемонстрировано в листинге 9.51 при взятии года и месяца из формы. Не забудьте, что `form['field']` возвращает, как правило, объект `MiniFieldStorage`, и значение находится в атрибуте `value`. Метод `getvalue()` возвращает значение (или список значений) напрямую.

Модуль `cgi` предлагает несколько функций низкого уровня. Следующие функции можно использовать для разбора полученных данных.

- `cgi.parse(fp) -> dict`

  Разбирает строку запроса (из переменной окружения, из файла `fp` или со стандартного ввода). Возвращает словарь вида `{'name': ['value1', 'value2', ...]}`.

- `cgi.parse_qs(qs[, keep_blank_values, strict_parsing]) -> dict`

  Разбирает строку `qs` (с содержимым, закодированным `x-www-form-urlencoded`). Флаг `keep_blank_values` индицирует необходимость сохранять пустые значения, а `strict_parsing` заставляет более критично относиться к ошибкам в строке запроса, возбуждая исключение `ValueError`.

- `cgi.parse_qsl(qs[, keep_blank_values, strict_parsing]) -> items`

  То же, что и `cgi.parse_qs`, но возвращает результат в виде списка кортежей `(имя, значение)`.

- `parse_multipart(fp, pdict)`

  Разбирает ввод в формате `multipart/form-data`. Аргумент `fp` — файл, из которого осуществляется ввод, а `pdict` — словарь с параметрами поля `content-type`.

- `cgi.parse_header(s) -> (value, dict)`

  Разбирает поле заголовка (например, `content-type`) на основное значение `value` и словарь с другими параметрами `dict`.

Следующие функции можно использовать для отладки CGI.

- `cgi.test()`

  Выводит на стандартный вывод всю доступную сценарию информацию в виде HTML.

- `cgi.print_environ()`

  Выводит информацию о переменных окружения.

- `cgi.print_environ_usage()`

  Выводит список полезных переменных окружения, которые применяются в CGI.

- `cgi.print_form(form)`

  Выводит содержимое формы `form`.

- `print_directory()`

  Выводит путь к текущему рабочему каталогу.

Описанная ниже функция может оказаться полезной для подготовки строки к включению в HTML-документ.

- `escape(s[, quote]) -> s1`

  Заменяет в строке `s` символы "`&`", "`<`" и "`>`" на их эквиваленты в HTML: `&`, `<`, `&gt`. Если аргумент `quote` задан и имеет истинное значение, то кавычки (") преобразуются в `"`.

В наших примерах имя появлялось в строке запроса только один раз. Если же одному имени соответствует несколько значений, атрибут `value` объекта

класса `FieldStorage` будет содержать список всех этих значений. Для корректной обработки подобных ситуаций нужно предусматривать проверку типа, например:

```
months = form["month"].value
if type(months) is type(""):
  # Одно значение делаем списком из одного элемента
  months = [months]
months = map(int, months)
# Теперь months содержит список чисел
```

Иногда сценарий используется для принятия файла. В листинге 9.52 приведен сценарий, который принимает файл и выводит его на Web-страницу.

Листинг 9.52. Файл upload.py. Сценарий, принимающий файл от HTTP-клиента

```
#!/usr/bin/env python
import cgi
print "Content-Type: text/html"
print
form = cgi.FieldStorage()
contents = "<b>Файл не загружен</b>"
try:
  fileitem = form["myfile"]
  if fileitem.file:
    # если файл, читаем содержимое файла целиком
    contents = "\n<PRE>%s</PRE>" % fileitem.file.read()
except:
  pass

print """<HTML><HEAD><TITLE>Прием файла</TITLE></HEAD>
<BODY><H1>Прием файла</H1><P>%s
<P><FORM ENCTYPE="multipart/form-data" ACTION="upload.py" METHOD="POST">
<br>Файл: <INPUT TYPE="file" NAME="myfile">
<br><INPUT TYPE="submit" NAME="b" VALUE="Отправить файл">
</FORM></BODY></HTML>""" % contents
```

Возможно, написать CGI-сценарий с первого раза правильно не удастся. Отладку CGI-сценария можно производить как с использованием HTTP-сервера, так и (в простых случаях) без него.

Возникающие в сценарии ошибки обычно пишутся в журнал (log) ошибок HTTP-сервера. Если смотреть журнал по каким-либо причинам неудобно, можно перенаправить вывод ошибок. Для этого документация по Python рекомендует добавить следующие строки к программе:

```
import sys, traceback
print "Content-Type: text/html"
print
sys.stderr = sys.stdout
try:
  # ... текст сценария ...
except:
  # выводим трассировочную информацию
  print "\n\n<PRE>"
  traceback.print_exc()
  print "\n</PRE>"
```

Примечание

Начиная с Python 2.1 отладку можно сделать еще проще, заменив функцию `sys.excepthook` на свою. *(См. разд. 9.2.1.)*

В простых случаях можно обойтись вообще без HTTP-сервера, добавив переменные окружения, которые обычно добавляет HTTP-сервер, запуская CGI-сценарий. Например:

Листинг 9.53. Файл cal-dbg.cgi. Отладка сценария `cal.cgi`

```
import os

os.environ.update({
  'REQUEST_METHOD': "GET",
  'QUERY_STRING': "year=2001&month=8",
  # и возможно другие
})
execfile("cal.cgi")
```

Правда, не стоит забывать, что в `QUERY_STRING` могут встречаться символы, которые потребуют кодирования. Для кодирования можно использовать функцию `urllib.quote_plus`, применяя ее к каждому имени и значению отдельно.

Следует заметить, что CGI-сценарии по своей среде выполнения могут сильно отличаться от обычных сценариев, запускаемых из командной стро-

ки. Это связано с тем, что они обычно имеют очень мало прав на чтение и запись файлов, переменные PATH и PYTHONPATH могут содержать минимальные списки путей, и т. д.

Кроме того, при разработке CGI-сценариев следует уделять большое внимание безопасности. Например, нельзя доверять ничему, что поступает извне: строки запроса можно запросто сфабриковать *(см. разд. 9.12.2 и 9.12.4)*. Поэтому следует внимательно смотреть, чтобы данные от пользователя не применялись напрямую в именах читаемых (и тем более — записываемых) файлов, а также текстуально не входили бы в команды, запускаемые по os.system и подобными методами.

Следующий пример показывает, как пользователь CGI-сценария может прочитать ему не предназначенный любой доступный для чтения файл. Подобная ситуация является нарушением безопасности, т. к. может дать злоумышленнику лишнюю информацию о системе.

Листинг 9.54. Файл insecure.cgi. Небезопасный CGI-сценарий

```
#!/usr/bin/env python
import cgi
print "Content-Type: text/html"
print
form = cgi.FieldStorage()

contents = ""
try:
  quarter = form["quarter"].value
  file = open("/home/httpd/reports/" + quarter, "r")
  contents = "\n<PRE>%s</PRE>" % file.read()
except:
  pass

print """<HTML><HEAD><TITLE>Отчет за квартал</TITLE></HEAD>
<BODY><H1>Отчет за квартал</H1><P>%s
<P><FORM ACTION="insecure.cgi">
<br>Квартал: <select name="quarter" size="1">
 <option value="1"> I <option value="2"> II
 <option value="3"> III <option value="4"> IV </select>
<br><INPUT TYPE="submit" NAME="b" VALUE="Получить отчет">
</FORM></BODY></HTML>""" % contents
```

Чтобы увидеть проблему, достаточно расположить приведенный сценарий на HTTP-сервере, условно названном `insecure.comp`, и ввести следующий URL:

```
http://insecure.comp/cgi-bin/insecure.cgi?quarter=/../../../etc/passwd
```

Так как в сценарии не сделано никаких проверок входной информации, а значение `quarter` непосредственно используется в формировании имени файла, стало возможным с помощью перехода к родительским каталогам обратиться к файлу /etc/passwd, традиционно содержащему информацию о пользователях.

Разумеется, приведенный сценарий не исчерпывает всех проблем с нарушением безопасности. К сожалению, надежная защита системы требует больших знаний и опыта, но соблюдать элементарные меры безопасности должен каждый программист.

9.12.2. Модуль *urllib*

Информационные пространства, море электронных документов совершенно бесполезны без универсальной адресации. Обычно пользователь передает браузеру или другой программе, осуществляющей связь между ним и Web-клиентом, адрес *URL (Uniform Resource Locator, универсальный указатель местонахождения ресурса)*. Передаваемый от Web-клиента Web-серверу *URI (Uniform Resource Identifier, универсальный идентификатор ресурса)* содержит информацию о местонахождении ресурса и строку запроса и является как бы координатой нужного ресурса в информационном пространстве сети. О структуре URI *см. разд. 9.12.3*, где описан модуль `urlparse`.

Модуль `urllib` позволяет работать с некоторыми типами URL, доступными в *WWW (World Wide Web, всемирная паутина)*: `http`, `https`, `ftp`, `gopher` и др. В модуле определены следующие функции.

- `urllib.urlopen(url[, data]) -> fobj`

  Открывает `url` и возвращает файлоподобный объект `fobj`, из которого можно считать соответствующий URL-документ. Если URL относится к типу `http` и представляет собой POST-запрос, параметр `data` должен содержать данные в формате `application/x-www-form-urlencoded`. См. функцию `urllib.urlencode()` ниже.

  Возвращаемый `urllib.urlopen()` объект класса `URLopener` имеет следующие методы: `read()`, `readline()`, `readlines()`, `fileno()`, `close()`, `info()` и `geturl()`. Заметьте, что методы для произвольного доступа, свойственные файлам, не поддерживаются. Метод `geturl()` дает окончательный URL выгружаемого из сети файла (с учетом всех переадресовок). Метод `info()` возвращает экземпляр класса `mimetools.Message`, в котором содержится дополнительная информация, связанная с URL. Для

URL `http`-типа в нее входит информация, передаваемая в заголовке ответа HTTP-сервера (`'content-type'`, `'content-length'` и т. п.).

Функция `urllib.urlopen()` прозрачно работает с прокси-серверами, не требующими аутентификации. В среде UNIX и Windows задать прокси можно, записав в переменные окружения `http_proxy`, `ftp_proxy` и `gopher_proxy` URL соответствующих прокси-серверов. Это можно сделать средствами операционной системы или в самом начале программы на Python:

```
import os, urllib
os.environ['http_proxy'] = "http://proxy.onego.ru:3128"
print urllib.urlopen("http://www.onego.ru").read()
```

- `urllib.urlretrieve(url[, fn[, hook[, data]]]) -> (filename, headers)`

Функция предназначена для выгрузки файлов из сети на локальный диск.

Открывает `url` и сохраняет его в файл с именем `fn`, возвращая имя файла `filename` и `headers` с данными метода `info()` (или `None`). При отсутствии `filename` генерируется имя вре́менного файла. Если URL относится к типу `http` и представляет собой POST-запрос, параметр `data` должен содержать данные в формате `application/x-www-form-urlencoded`. Через параметр `hook` передается функция, которая имеет три параметра: количество переданных к данному моменту блоков, размер блока в байтах и полный размер передаваемых данных в байтах (или `-1`). С помощью `hook()` можно, например, организовать индикатор состояния загрузки URL.

- `urllib.urlcleanup()`

Очищает кэш, который мог остаться после вызова `urlretrieve()`.

- `urllib.quote(s[, safe]) -> s1`
  `urllib.quote_plus(s[, safe]) -> s1`

Экранирует некоторые символы строки `s`, записывая их в виде `"%xx"`. Буквы, цифры, символы `"_"`, `","`, `"."` и `"-"`, а также символы строки `safe` (по умолчанию `"/"`), не экранируются. Функция `urllib.quote_plus()` аналогична `urllib.quote()`, но заменяет пробелы на знаки `"+"`.

- `urllib.unquote(s) -> s1`
  `urllib.unquote_plus(s) -> s1`

Функции, обратные `urllib.quote()` и `urllib.quote_plus()` соответственно.

Примеры:

```
>>> import urllib
>>> text = "Текст для пробы."
```

```
>>> print urllib.quote(text)
%f4%c5%cb%d3%d4%20%c4%cc%d1%20%d0%d2%cf%c2%d9.
>>> print urllib.quote_plus(text)
%f4%c5%cb%d3%d4+%c4%cc%d1+%d0%d2%cf%c2%d9.
>>> print urllib.unquote(urllib.quote(text))
Текст для пробы.
```

- `urllib.urlencode(query[, doseq]) -> s`

  Преобразует `query` (словарь или список двухэлементных кортежей имя-значение) в строку, которую можно передавать функции `urllib.urlopen()` в качестве параметра `data`. Если необязательный параметр `doseq` имеет значение "истина", для значений, являющихся списками, `имя=значение` генерируется для каждого элемента списка.

  Следующий пример показывает работу функции `urllib.urlencode()`:

```
>>> from urllib import urlencode
>>> print urlencode({"a": 1, "b": "(текст)"})
b=%28%F2%E5%EA%F1%F2%29&a=1
>>> print urlencode({"a": 1, "b": [1, 1, 0, 1]}, 1)
b=1&b=1&b=0&b=1&a=1
>>> print urlencode([("a", 1), ("b", "(текст)")])
a=1&b=%28%F2%E5%EA%F1%F2%29
>>> print urlencode([("a", 1), ("b", [1, 1, 0, 1])], 1)
a=1&b=1&b=1&b=0&b=1
>>> print urlencode([("a", 1), ("b", [1, 1, 0, 1])], 0)
a=1&b=%5B1%2C+1%2C+0%2C+1%5D
```

В листинге 9.55 представлена функция `make_req()` для запроса к HTTP-серверу с использованием методов `GET` и `POST`.

Листинг 9.55. Файл make_req.py. Запрос к HTTP-серверу

```
import urllib
def make_req(url, query, method="GET"):
  """make_req(url, query[, method]) -> page
  функция для возвращает результат запроса query к url по методу
  method"""
  querystring = urllib.urlencode(query)
  if method == "GET":
    f = urllib.urlopen(url + "?" + querystring)  # метод GET
```

```
  else:
    f = urllib.urlopen(url, querystring)  # метод POST
  return f.read()

# поиск на Google
print make_req("http://google.com/search",
   {"hl": "en", "safe": "off", "q": "Python"})
# поиск в базе данных программного обеспечения для Python
print make_req("http://www.vex.net/parnassus/apyllo.py",
   {"find": "Oracle"}, "POST")
```

Примечание

Возможно, что приведенные в листинге 9.55 примеры не сработают. Скорее всего, из-за смены URL или названий полей HTML-форм. Вы можете легко найти в сети другие примеры для запросов. Практически везде, где на Web-странице заполняется некоторая Web-форма, применяются методы `GET` или `POST`.

В Python 2.0 и выше появился также модуль `urllib2`, который мы здесь не рассматриваем. Этот модуль имеет более широкие, чем `urllib` возможности. Например, поддерживается аутентификация на HTTP-сервере и безопасные соединения (HTTPS).

9.12.3. Модуль *urlparse*

Модуль `urlparse`, как и подразумевает его название, служит для разбивки универсальных идентификаторов ресурсов (URI) на компоненты и обратной операции, а также для объединения базового и относительного URL в абсолютный. Точное определение URI заинтересованный читатель может найти в RFC-2396 и RFC-2616.

Функция urlparse.

- `urlparse.urlparse(url[, def_scheme[, allow_frags]]) -> tuple6`

  где `url` — строка с URL; `def_scheme` — адресная схема по умолчанию; `allow_frags` — флаг, указывающий на необходимость выделять фрагмент при разборе; `tuple6` — результат в виде шестиэлементного кортежа: `(scheme, netloc, path, params, query, frag)`; `scheme` — адресная схема, например, `"http"`; `netloc` — местонахождение в сети; `path` — путь к ресурсу; `params` — параметры; `query` — строка запроса; `frag` — идентификатор фрагмента.

  Функция предполагает, что URL/URI строится по следующему шаблону:

  ```
  scheme://netloc/path;parameters?query#fragment
  ```

Любой из компонентов может отсутствовать (в этом случае в кортеж записываются пустые строки). Разделители не принимаются во внимание, кроме, может быть начальной косой черты "/" в компоненте path.

```
>>> from urlparse import *
>>> URL = "http://google.com/search?q=Tkinter"
>>> print urlparse(URL)
('http', 'google.com', '/search', '', 'q=Tkinter', '')
>>> print urlunparse(
...     ('http', 'google.com', '/search', '', 'q=Tkinter', ''))
http://google.com/search?q=Tkinter
```

- `urlparse.urlunparse(tuple6) -> url-string`

Операция, обратная `urlparse.urlparse()`. Функция возвращает строку с URL, соответствующую кортежу `tuple6`. Возвращенное значение необязательно совпадает с исходным URL.

Указатель местонахождения ресурса, URL, помещенный в документ в качестве ссылки, может быть не только абсолютным, но и относительным. Такой URL не содержит имени сервера, а начинается сразу с элемента path или его части. Предполагается, что запрос по такому URL будет сделан на тот сервер, с которого было получено содержание, идентифицированное с помощью URI. Следующая функция позволяет соединить базовый и относительный указатели ресурса.

- `urlparse.urljoin(base, url[, allow_frags]) -> abs_url`

где `base` — базовый URL; `url` — относительный (для данного базового) URL; `allow_frags` — см. выше; `abs_url` — результирующий абсолютный указатель ресурса.

Эта функция соединяет базовый и относительный идентификаторы ресурсов в один абсолютный указатель. Например:

```
>>> import urlparse
>>> base = "http://server/path/page.html#1"
>>> url = "../index.html#2"
>>> print urlparse.urljoin(base, url, 1)
http://server/index.html#2
```

Программа в листинге 9.56 заменяет в HTML странице с заданным URL относительные ссылки на абсолютные и печатает результат на стандартный вывод.

Листинг 9.56. Замена относительных ссылок на абсолютные

```
import re, string, urlparse, urllib
global base_url
```

```
base_url = "http://www.python.org"  # базовый URL
# словарь тегов и атрибутов, в которых задается URL
tags = {
"A": "href",              "HEAD": "profile",
"APPLET": "codebase",     "IFRAME": "longdesc|src",
"AREA": "href",           "IMG": "longdesc|src|usemap",
"BASE": "href",           "INPUT": "src|usemap",
"BLOCKQUOTE": "cite",     "INS": "cite",
"BODY": "background",     "LINK": "href",
"DEL": "cite",            "OBJECT": "classid|codebase|data|usemap",
"FORM": "action",         "Q": "cite",
"FRAME": "longdesc|src", "SCRIPT": "src",
}

# подготовка регулярного выражения для поиска тегов
TAGS = string.join(tags.keys(), "|")
tag_re = re.compile("(\<\s*)(%s)(\s+[^>]+?|)(\>)" % TAGS, re.I|re.M)

def replace_attr(m):
  """Замена значения атрибута, содержащего URL"""
  global base_url
  attr = list(m.group(1, 2, 3, 4, 5))
  attr[3] = urlparse.urljoin(base_url, attr[3])
  return string.join(attr, "")

def replace_tag(m):
  """Поиск тегов, возможно, содержащих URL в одном из атрибутов,
  и замена значений в этих атрибутах"""
  lt, tag, atts, gt = m.group(1, 2, 3, 4)
  ATTS = tags[string.upper(tag)]
  attr_re = """(\s+)(%s)(\s*=\s*['"]?)([^'" \n\t\r]+)(['"]?)""" % ATTS
  return lt + tag + re.sub(attr_re, replace_attr, atts, re.I) + gt

page = urllib.urlopen(base_url).read()  # получение и
print tag_re.sub(replace_tag, page)     # печать HTML-страницы
```

9.12.4. Модуль *httplib*

Описываемый модуль `httplib` определяет классы, реализующие клиентов протокола HTTP. Это более низкоуровневый доступ к протоколу, чем реализованный модулем `urllib`, и потому может использоваться для не совсем

стандартных нужд (например, испытания Web-сервера). Позволяет устанавливать не только обычные, но и безопасные SSL-соединения. Модуль был серьезно переработан при переходе от версии 1.5.2 к 2.0 и теперь поддерживает HTTP 1.1.

Ниже описан класс `HTTP` (общий для версий Python 1.5 и 2). Если необходимо более детальное изучение модуля, можно обратиться к исходному тексту модуля (см. файл python2.0/httplib.py на компакт-диске).

Модуль предоставляет класс `HTTP`, экземпляр которого соответствует одному соединению с HTTP-сервером.

```
httplib.HTTP([host[, port]]) -> httpobj
```

где: `host` — строка, адрес сервера (возможно, с портом через двоеточие). Если этот параметр не указан, соединение не устанавливается при инициализации объекта; `port` — номер порта. По умолчанию используется 80; `httpobj` — экземпляр класса HTTP, заведующий соединением с HTTP-сервером.

Методы объектов класса `HTTP` приведены в табл. 9.71.

Таблица 9.71. *Методы объектов класса* `HTTP`

Метод	Описание
`connect([host [, port]])`	Устанавливает соединение. Аргументы те же, что и у конструктора класса. Требуется тогда, когда адрес сервера не был задан при вызове конструктора
`endheaders()`	Посылает пустую строку как признак окончания заголовочной части
`getfile() -> file`	Получает файловый объект, из которого можно прочитать возвращенный сервером ответ на запрос (обычно текст HTML-страницы)
`getreply() -> (code, reason, msg)`	Получает ответ сервера в виде кортежа. Значение `code` — целое число, обозначающее стандартный код ответа (обычно `200`), `reason` — ответ в текстовом виде (например, `"OK"` или `"Not found"`), `msg` — объект класса `mimetools.Message` с заголовками ответа
`putheader(header, arg[, ...])`	Отправляет на сервер отдельное поле заголовка с именем `header` и значением в `arg`. При задании более одного аргумента, используются строки продолжения. (Спецификация RFC-822)
`putrequest(request, url)`	Посылает запрос на HTTP-сервер. Параметр `request` указывает тип запроса (например, `"GET"`), а `url` — запрашиваемый с сервера URL (обычно относительно данного сервера, например `/index.html`)

Таблица 9.71 *(окончание)*

Метод	Описание
`send(data)`	Посылает данные `data` на сервер
`set_debuglevel(level)`	Настраивает уровень печатаемых отладочных сообщений: `0` — отсутствие отладки

Приведенные методы можно использовать только в строго определенной последовательности:

1. Соединиться с HTTP-сервером с помощью метода `connect()` (если это не было сделано при инициализации объекта).
2. Ровно один раз вызвать метод `putrequest()`.
3. Сделать ноль или более вызовов метода `putheader()`.
4. Вызвать метод `endheaders()`.
5. Необязательные вызовы метода `send()` для отправки данных на сервер.
6. Вызвать метод `getreply()`.
7. Вызвать метод `getfile()` и считать данные из возвращенного им файлового объекта.

Пример в листинге 9.57 демонстрирует применение модуля `httplib`. Поисковый сервер **http://google.com/** выдает различные страницы в зависимости от запрашиваемого языка.

Листинг 9.57. Демонстрация работы модуля `httplib`

```
from httplib import HTTP
conn = HTTP("google.com")     # http://google.com/
conn.putrequest("GET", "/")
conn.putheader("accept-language", "ru")  # русский язык
conn.endheaders()
(code, reason, msg) = conn.getreply()
page = conn.getfile().read()
print code, reason, msg, page
```

В версиях Python 2.*x* модуль `httplib` предоставляет классы для работы с протоколом HTTP/1.1. Эти классы называются `httplib.HTTPConnection` (HTTP-соединение), `httplib.HTTPSConnection` (безопасное HTTP-соединение) и `httplib.HTTPResponse` (HTTP-отклик). Таким образом, в новой версии модуля запросы к серверу и отклики сервера разведены в разные классы. Это позволяет немного распараллеливать отправку запросов и полу-

чение ответов. В документации приводится диаграмма состояний (рис. 9.2), которые может принимать объект `HTTPConnection`. На этой же диаграмме видны методы объектов этого класса.

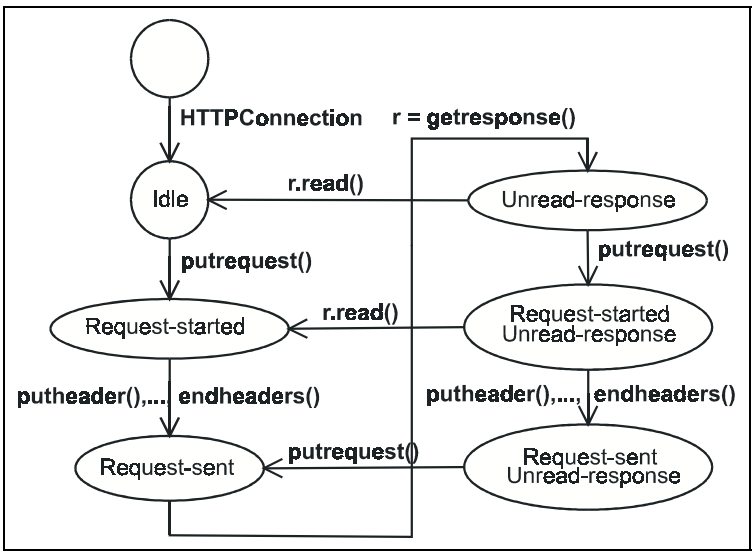


Рис. 9.2. Диаграмма состояний объекта класса `HTTPConnection`

Объекты класса `HTTPResponse` содержат ответ сервера на отправленный ему запрос. В табл. 9.72 приведены некоторые атрибуты объектов этого класса.

Таблица 9.72. *Некоторые атрибуты методов класса* `HTTPResponse`

Атрибут	Описание
`fp`	Файловый объект, из которого можно считать тело ответа сервера
`msg`	Объект `mimetools.Message`, содержащий заголовочную часть ответа сервера
`status`	Код ответа
`reason`	Объяснение причины ответа
`version`	Версия протокола (`11` для HTTP/1.1)
`length`	Длина ответа

В листинге 9.58 приведен тот же пример, что и выше, но с использованием класса `HTTPConnection` и объектов класса `HTTPResponse`.

Листинг 9.58. Демонстрация работы модуля `httplib` (вариант 2)

```
from httplib import HTTPConnection
conn = HTTPConnection("google.com")
conn.putrequest("GET", "/")
conn.putheader("accept-language", "ru")
conn.endheaders()
resp = conn.getresponse()
page = resp.fp.read()
print resp.status, resp.reason, resp.msg, page
```

9.12.5. Модуль *smtplib*

Модуль `smtplib` может использоваться для организации работы по *SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты)*. Протоколы SMTP и ESMTP (расширенный SMTP) описаны в RFC-821 и RFC-1869. Модуль определяет класс `smtplib.SMTP` для управления SMTP-соединением (методы его объектов даны в табл. 9.73) и множество исключений, которые описаны в табл. 9.74. Модуль `smtplib` предназначен для работы в клиентской части SMTP.

```
smtplib.SMTP([host[, port]]) -> smtp_session_obj
```

Конструктор класса `smtplib.SMTP`. Параметры `host` и `port` указывают SMTP-сервер, к которому необходимо подключиться. Для `port` по умолчанию используется стандартный для SMTP порт 25. Если `host` и `port` заданы, конструктор сам установит соединение, в противном случае придется вызывать метод `connect()` отдельно. Хотя объекты класса `SMTP` имеют методы практически для любой команды SMTP-протокола, для типичной отправки почты достаточно конструктора и высокоуровневых методов `sendmail()` и `quit()`.

***Таблица 9.73.** Методы объектов класса `smtplib.SMTP`*

Метод	Описание
`set_debuglevel(level)`	Устанавливает уровень отладочной информации. Ненулевое значение включает вывод отладочных сообщений
`connect([host [, port]])`	Устанавливает соединение с данным хостом и портом. По умолчанию `host` — `localhost`, а `port` — 25. Данный метод вызывается конструктором, если ему были переданы оба параметра

Таблица 9.73 *(окончание)*

Метод	Описание
`docmd(cmd, [, arg]) -> (rcode, rtext)`	Выполняет команду `cmd` с аргументом `arg` на SMTP-сервере и возвращает кортеж из `rcode` (цифровой код результата) и `rtext` (текст ответа сервера)
`helo([hostname]) -> (rcode, rtext)`	Клиент представляется серверу с использованием имени `hostname`. Возвращает кортеж с кодом и текстом ответа сервера
`ehlo([hostname]) -> (rcode, rtext)`	Аналог `smtplib.helo()` для ESMTP-сервера
`has_extn(name) -> 0, 1`	Проверяет поддержку ESMTP расширения с именем `name`
`verify(address) -> (rcode, rtext)`	Проверяет адресат `address` по команде `VRFY` протокола SMTP. Многие сайты отключают эту возможность
`sendmail(...) -> {...}`	Метод для отправки почты. Описан отдельно
`quit()`	Завершает SMTP-сессию и закрывает соединение

Кроме перечисленных методов, класс `smtplib.SMTP` предоставляет низкоуровневые методы `helo()`, `ehlo()`, `expn()`, `help()`, `mail()`, `rcpt()`, `vrfy()`, `send()`, `noop()`, и `data()`, соответствующие командам SMTP-протокола. Интересующиеся могут посмотреть исходный текст модуля `smtplib`, чтобы понять, как эти команды используются.

Самый важный метод в объекте класса `smtplib.SMTP` — `sendmail()`. Он принимает следующие аргументы:

```
sendmail(from_addr, to_addrs, msg[, mail_opt, rcpt_opt]) -> dict
```

где `from_addr` — строка с полным адресом отправителя; `to_addrs` — строка с адресом получателя или список адресов получателей; `msg` — отправляемое сообщение (включая заголовочную часть, отделенную от тела сообщения пустой строкой); `mail_opt` — список опций для команды `"MAIL FROM"`; `rcpt_opt` — список опций для команды `RCPT`; `dict` — словарь, ключами которого являются адреса отклоненных адресатов, а значения — ответы сервера на эти адреса. Если доставка прошла успешно, этот словарь пуст.

При работе метода `sendmail()` могут возникнуть исключения (табл. 9.74).

Таблица 9.74. *Исключения, описанные в модуле* `smtplib`

Исключение	Описание
`smtplib.SMTPException`	Базовый класс для всех исключений модуля
`smtplib.SMTPServerDisconnected`	Сервер неожиданно прервал связь или связь с сервером еще не была установлена

Таблица 9.74 *(окончание)*

Исключение	Описание
smtplib.SMTPResponseException	Базовый класс для всех исключений, которые содержат код ответа SMTP-сервера
smtplib.SMTPSenderRefused	Отправитель отвергнут
smtplib.SMTPRecipientsRefused	Все получатели отвергнуты сервером
smtplib.SMTPDataError	Сервер ответил неожиданным кодом на данные сообщения
smtplib.SMTPConnectError	Ошибка установления соединения
smtplib.SMTPHeloError	Сервер не ответил правильно на команду HELO или отклонил ее

В листинге 9.59 показано, как с помощью модуля smtplib отправить сообщение.

Листинг 9.59. Файл send_mail.py. Отправка сообщения

```
from smtplib import SMTP
fromaddr = "book_reader@mail.ru"  # от кого
toaddr = "rnd@onego.ru"           # кому
message = """From: Book Reader <book_reader@mail.ru>
To: Book Writer <rnd@onego.ru>
Subject: Python

Здравствуйте! Я читаю книгу "Python в подлиннике" и
запустил программу send_mail.py.
"""
s = SMTP('mail.onego.ru')
s.set_debuglevel(1)
s.sendmail(fromaddr, toaddr, message)
s.quit()
```

9.12.6. Модуль *poplib*

Модуль poplib предоставляет сервис по доступу к почтовому серверу по протоколу POP3 (определен в RFC-1725). Модуль определяет класс POP3, отвечающий за сеанс работы с POP3-сервером.

```
poplib.POP3(host[, port]) -> popobj
```

где `host` — адрес POP3-сервера; `port` — порт на сервере (по умолчанию 110); `popobj` — объект для управления сеансом работы с POP3-сервером.

При работе объект класса `POP3` может возбуждать исключения `poplib.error_proto`.

В табл. 9.75 приведены методы объектов класса `POP3`.

Таблица 9.75. *Методы объектов класса* `POP3`

Метод	Команда POP3	Описание
`getwelcome() -> s`		Возвращает строку с приветствием POP3-сервера
`user(username) -> s`	`USER username`	Посылает команду `USER` с указанием имени пользователя `username`. Возвращает строку с ответом сервера
`pass_(password) -> s`	`PASS password`	Посылает пароль пользователя по команде `PASS`. После этой команды и до команды `QUIT` почтовый ящик блокируется
`apop(user, secret) -> s`	`APOP user secret`	Другой способ аутентификации на сервере: APOP
`rpop(user) -> s`	`RPOP user`	Аутентификация по методу RPOP
`stat() -> (messages, size)`	`STAT`	Возвращает кортеж с информацией о почтовом ящике. `messages` — количество сообщений, `size` — размер почтового ящика в байтах
`list([num]) -> res`	`LIST [num]`	Возвращает список сообщений в формате `(response, ['mesg_num octets', ...])` (если не указан `num`) и `"+OK num octets"` (если указан). Здесь список `lst` состоит из строк в формате `"num octets"`
`retr(num) -> (response, lst, octets)`	`RETR num`	Извлекает сообщение с номером `num`
`dele(num) -> response`	`DELE num`	Удаляет сообщение с номером `num`
`rset() -> response`	`RSET`	Убирает пометки на удаление сообщений
`noop() -> response`	`NOOP`	Ничего не делает. Используется для поддержания соединения
`quit() -> response`	`QUIT`	Отключает от сервера. Сервер выполняет все необходимые изменения и снимает блокировку с почтового ящика

Таблица 9.75 *(окончание)*

Метод	Команда POP3	Описание
`top(num, lines) -> (response, lst, octets)`	`TOP num lines`	Аналогичен `retr()`, но возвращает только `lines` строк тела сообщения (заголовочная часть возвращается полностью)
`uidl([num]) -> res`	`UIDL [num]`	Сокращение от unique-id listing, возвращает уникальные идентификаторы сообщений. Формат `(response, lst, octets)`, если `num` не указан, и `"+OK num uniqid"`, если указан

В табл. 9.75 под `num` понимается номер сообщения (номер конкретного сообщения не меняется на протяжении всей сессии), `response` — ответ сервера, начинающийся с `"+OK "` для успешных операций (при безуспешных возбуждается исключение `poplib.proto_error`). Параметр `octets` — количество байтов в сообщении или его принятых данных. `uniqid` — идентификатор сообщения, порождаемый сервером (строка из 1—70 символов с кодами от `0x21` до `0x7E`).

Работа с POP3-сервером состоит из трех этапов: аутентификации, транзакций и обновления. На этапе аутентификации, наступающем сразу после создания POP3-объекта, подаются команды `USER`, `PASS`, а в некоторых случаях и `APOP`, `RPOP`. После аутентификации POP3-серверу становится ясно, какой именно пользователь к нему подключился, и наступает этап транзакций. Здесь можно подавать все остальные команды. Этап обновления вызывается командой `QUIT`, по которой POP3-сервер обновляет почтовый ящик пользователя в соответствии с выполненными транзакциями: удаляет сообщения, помеченные для удаления.

В листинге 9.60 программа на Python подключается к POP3-серверу бесплатной почтовой службы **mail.ru**, печатает отправителя, получателя и тему, принимает и удаляет самое длинное сообщение. Для иллюстрации, выполняются и другие команды.

Листинг 9.60. Файл book_reader.py. Доступ к POP3-ящику

```
import poplib, string, rfc822, StringIO
p = poplib.POP3("pop.mail.ru")
print p.getwelcome()
# этап аутентификации
print p.user("book_reader")
print p.pass_("secret")
```

```
# этап транзакций
response, lst, octets = p.list()
print response
mxnum, mxsize = 0, 0
for msgnum, msgsize in map(string.split, lst):
  # print msgnum, msgsize
  if int(msgsize) > mxsize:
    mxnum, mxsize = int(msgnum), int(msgsize)
if mxsize:
  print "Сообщение %(mxnum)s самое длинное: %(mxsize)s байт" % vars()
  print "UIDL =", string.split(p.uidl(mxnum))[2]
  (resp, lines, octets) = p.top(mxnum, 0)
  msgtxt = string.join(lines, "\n")+"\n\n"
  msg = rfc822.Message(StringIO.StringIO(msgtxt))
  print "Удаляем сообщение:"
  print "* От: %(from)s\n* Кому: %(to)s\n* Тема: %(subject)s\n" % msg
  print p.dele(mxnum)
else:
  print "Ящик пуст."
# этап обновления
print p.quit()
```

Результат выполнения:

```
+OK mPOP POP3 server ready <41875.999069178@mail.ru>
+OK Password required for user book_reader
+OK book_reader@mail.ru maildrop has 4 messages (5177 octets)
+OK 4 messages (5177 octets)
Сообщение 3 самое длинное: 1622 байт
UIDL = 99906915527744
Удаляем сообщение:
* От: Roman Suzi <rnd@onego.ru>
* Кому: <book_reader@mail.ru>
* Тема: =?koi8-r?B?0NLPws7PxSDTz8/C3cXOycUgKGZ3ZCk=?=

+OK message 3 deleted
+OK POP3 server at mail.ru signing off
```

Обратите внимание, что тема письма закодирована base64, и кодировка букв — koi8-r.

9.12.7. Модуль *telnetlib*

Посредством модуля `telnetlib` можно программировать многое, что доступно с помощью telnet (а это немало, если учесть, что telnet можно использовать для большинства распространенных в Internet протоколов, если, конечно, знать, что эти протоколы требуют). Следующий пример показывает, как получить точное время с сервера `time.nist.gov`:

```
>>> tn = telnetlib.Telnet("time.nist.gov", 13)
>>> print tn.read_all()

52124 01-08-03 19:16:58 50 0 0 163.2 UTC(NIST) *
```

Это аналогично использованию команды `telnet`:

```
$ telnet time.nist.gov 13
Trying 192.43.244.18...
Connected to time.nist.gov (192.43.244.18).
Escape character is '^]'.

52124 01-08-03 19:16:58 50 0 0 163.2 UTC(NIST) *
Connection closed by foreign host.
```

Работа с модулем начинается с создания экземпляра класса `Telnet`. Конструктор класса `telnetlib.Telnet` описан ниже.

```
telnetlib.Telnet(host, port) -> tn
```

где `host` — адрес хоста, к которому производится подключение; `port` — порт на этом хосте; `tn` — экземпляр класса `Telnet`.

В табл. 9.76 представлены некоторые методы, которые можно использовать в объектах класса `Telnet`. Клиент telnet ведет две очереди: сырую (raw) и обработанную (cooked), в связи с чем возникает несколько вариантов метода `read()` от `read_very_lazy()` (читать очень лениво) до `read_very_eager()` (читать очень охотно).

Таблица 9.76. *Методы объектов класса* `Telnet`

Метод	Описание
`close()`	Закрывает соединение
`expect(list [, timeout])`	Читает, пока не будет встречено одно из регулярных выражений из списка `list`. Необязательный параметр `timeout` (тайм-аут) задается в секундах. Если он `None` или отсутствует, тайм-аут не используется

Таблица 9.76 *(окончание)*

Метод	Описание
`fileno()`	Возвращает файловый дескриптор используемого сокета
`get_socket()`	Возвращает используемый внутри сокет-объект
`interact()`	Функция для организации простейшего терминала
`mt_interact()`	Многопоточная версия `interact()`
`read_all()`	Читает все данные до конца. Возвращается при разрыве соединения
`read_eager()`	Читает готовые данные. Возбуждает исключение `EOFError`, если соединение закрыто, а данных в обработанной очереди нет. Возвращает "", если нет обработанных данных. Вызов отрабатывает без блокировок, если прием не застревает на середине IAC-последовательности
`read_lazy()`	Обрабатывает и читает данные из очередей. Возбуждает исключение `EOFError`, если соединение закрыто, а данных в обработанной очереди нет. Возвращает "", если нет данных. Вызов отрабатывает без блокировок, если прием не застревает на середине IAC-последовательности
`read_some()`	Читает по крайней мере один байт готовых данных, если не достигнут конец файла. Возвращает "" при конце файла. Блокируется до получения хотя бы одного байта
`read_until(expected [, timeout])`	Читает до строки `expected` или до наступления таймаута, в этом случае возвращается все, что было прочитано. Параметр `timeout` задается в секундах. В случае разрыва соединения возбуждается исключение `EOFError`
`read_very_eager()`	Читает все, что можно прочитать без блокировок. Возбуждает исключение `EOFError`, если соединение закрыто, а данных в обработанной очереди нет. Возвращает "", если нет данных. Вызов отрабатывает без блокировок, если прием не застревает на середине IAC-последовательности
`read_very_lazy()`	Читает данные из обработанной очереди. Возбуждает исключение `EOFError`, если соединение закрыто, а данных нет. Возвращает "", если нет данных. Никогда не блокируется
`set_debuglevel (level)`	Устанавливает уровень выводимой на стандартный вывод отладочной информации. Чем больше значение `level`, тем больше отладочной информации выводится
`sock_avail()`	Проверяет наличие данных в сокете
`write(s)`	Записывает данные в сокет, удваивая IAC-символы

Примечание

IAC (Interpret as Command, интерпретировать как команду) — управляющий символ с кодом 255, после которого следует команда.

9.12.8. Модуль *socket*

Модуль `socket` предоставляет доступ из Python к *сокетам*. Сокетом называется конечная точка сетевых коммуникаций. Сокеты используют транспортный уровень, согласно семиуровневой модели OSI (Open Systems Interconnection, взаимодействие открытых систем), т. е. относятся к более низкому уровню, чем большинство описываемых в этом разделе протоколов.

Каждый сокет относится к одному из *коммуникационных доменов*. Например, модуль `socket` поддерживает следующие домены: `UNIX` и `Internet`. Каждый домен подразумевает свое семейство протоколов и адресацию. Ниже мы будем говорить лишь о наиболее значимом сегодня домене `Internet`, в котором используются протоколы TCP/IP и UDP/IP.

Для создания соединения TCP/IP необходимо два сокета: один на локальной машине, а другой на удаленной. Таким образом, каждое сетевое соединение характеризуется IP-адресом локальной машины и портом на локальной машине, IP-адресом и портом на удаленной машине.

При реализации модели "клиент-сервер", сервер обычно слушает (listen) порт с определенным номером. Дождавшись запроса от клиента на этот порт, сервер и клиент устанавливают соединение, используя свободные порты. Данные по наиболее популярным портам приведены в табл. 9.77.

Таблица 9.77. *Некоторые общепринятые порты TCP и соответствующие сетевые сервисы*

Номер порта	Название порта	Сетевой сервис
21	`ftp`	FTP, File Transfer Protocol, протокол передачи файлов
23	`telnet`	Telnet
25	`smtp`	SMTP, Simple Mail Transfer Protocol, простой протокол передачи почты
53	`domain`	DNS, Domain Name Server, сервер имен
80	`www`	HTTP, Hyper Text Transfer Protocol, протокол передачи гипертекста
110	`pop3`	POP3, Post Office Protocol, протокол почтового отделения, версия 3 (для получения почты)

```
socket.socket(af, type[, proto]) -> sock_obj
```

Создает сокет из домена (адресной семьи) `af` (возможные значения: `socket.AF_UNIX` и `AF_INET` (табл. 9.78) типа `type` (см. названия констант в табл. 9.79). Необязательный параметр `proto` (протокол) обычно равен 0 (табл. 9.80) и имеет смысл только для сокетов типа `SOCK_RAW`.

Таблица 9.78. *Некоторые адресные семьи*

Константа	Описание
`AF_UNIX`	Домен UNIX
`AF_INET`	Домен Internet
`AF_INET6`	Internet нового поколения
`AF_APPLETALK`	Домен AppleTalk
`AF_IPX`	Домен IPX

Таблица 9.79. *Типы сокетов*

Тип сокета	Описание
`SOCK_STREAM`	Последовательный гарантированный двусторонний поток байтов с поддерживаемым соединением
`SOCK_DGRAM`	Непоследовательный негарантированный двусторонний поток данных в виде дейтаграмм фиксированной максимальной длины, без поддержания соединения
`SOCK_RAW`	Сырой сокет, дающий доступ к нижележащим протоколам
`SOCK_RDM`	Протокол надежной доставки сообщения
`SOCK_SEQPACKET`	Последовательный гарантированный двусторонний поток дейтаграмм фиксированной максимальной длины с поддержанием соединения

Примечание

Из указанных протоколов обычно используются только `SOCK_STREAM` и `SOCK_DGRAM`.

Таблица 9.80. *Некоторые типы протоколов*

Тип протокола	Номер	Константа	Описание
`ip`	0	`IPPROTO_IP`	IP, Internet Protocol, а также незначащий псевдономер

Таблица 9.80 *(окончание)*

Тип протокола	Номер	Константа	Описание
icmp	1	IPPROTO_ICMP	ICMP, Internet Control Message Protocol, протокол управляющих сообщений (например, для поддержки работы команды ping)
tcp	6	IPPROTO_TCP	TCP, Transmission Control Protocol, протокол управления передачей
udp	17	IPPROTO_UDP	UDP, User Datagram Protocol, протокол пользовательских дейтаграмм
ddp	37		DDP, Datagram Delivery Protocol, протокол доставки дейтаграмм
ipv6	41		IPv6 — Internet-протокол следующего поколения

Дальнейшая работа с сокет-объектом проводится вызовом его методов. Методы приведены в табл. 9.81. Далее параметр или возвращаемое значение address соответствуют адресной информации, принятой в данной адресной семье. Для сокетов, работающих на основе IP, адрес — пара (host, port), где host указывает IP-адрес хоста, а port — порт.

Таблица 9.81. *Методы сокет-объекта*

Метод	Описание
bind(address)	Привязывает сокет к локальному адресу address
accept() -> (socket_obj, address)	Ожидает входящего соединения и возвращает новый сокет-объект socket_obj и адресную информацию о клиенте address
close()	Закрывает сокет
connect(address)	Соединяет сокет с удаленным адресом address. В случае неудачи возбуждается исключение socket.error
connect_ex(address) -> errno	Аналогичен connect(), но возвращает код ошибки в случае неудачи
dup() -> sobj	Возвращает новый сокет-объект, соединенный с тем же системным ресурсом
fileno() -> fd	Возвращает файловый дескриптор сокета

Таблица 9.81 *(продолжение)*

Метод	Описание
`getpeername() -> address`	Дает адресную информацию об удаленной точке соединения
`getsockname() -> address`	Дает адресную информацию о локальной точке соединения
`getsockopt(level, option[, buffersize]) -> value`	Возвращает опцию сокета. Параметр `level` (уровень) задается одной из `SOL_`-констант, например `SOL_SOCKET`. Параметр `option` указывает требуемую опцию и, например, может быть одной из `SO_`-констант, такой как `SO_REUSEADDR`. (Эти детали лучше уточнить по документации к конкретной системе). Если параметр `buflen` (длина буфера) задан, то значение опции будет получено в строке длины `buflen`. Отсутствие `bufle` подразумевает целочисленное значение опции.
`listen(num)`	Заставляет сокет-объект ожидать ("слушать") соединения на сокете. Параметр `num` — число непринятых (по `accept()`) соединений в очереди, которые система позволит перед тем, как начнет отказывать в новых соединениях
`makefile(mode='r', bufsize=-1) -> file`	Возвращает файловый объект, ассоциированный с сокетом. Параметр `mode` по умолчанию `"r"`, а `bufsize` — `-1` (отсутствие буферизации). Файловый объект и сокет-объекты могут быть закрыты и удалены независимо друг от друга
`recv(bufsize [, flags]) -> data`	Получает данные из сокета. Параметр `bufsize` определяет максимальный объем порции данных, получаемых за один вызов `recv()`. Флаги составляются из `MSG_`-констант и их назначение лучше уточнить в документации к конкретной ОС. Например, указание `MSG_PEEK` позволяет прочитать данные из очереди приема, не убирая их из очереди
`recvfrom(bufsize [, flags]) -> (data, address)`	Аналогичен `recv()`, но кроме данных возвращает адресную информацию посылающей стороны
`send(data[, flags]) -> bytes`	Посылает данные из строки `data` в сокет. Параметр `flags` имеет сходное с аналогичным параметром метода `recv()` значение. Возвращает количество переданных байтов
`sendto(string [, flags], address)`	Аналогичен `send()`, но производит отправку по адресу `address`. Сокет не должен быть уже соединен с удаленным хостом

Таблица 9.81 *(окончание)*

Метод	Описание
`setblocking(flag)`	Устанавливает режим блокировки сокета. Если `flag` равен `0`, сокет находится в неблокирующем режиме, иначе — в блокирующем (как и все вновь созданные сокеты). В неблокирующем режиме отсутствие данных в очереди при вызове `recv` или невозможность отправить данные по `send` приводят к исключению `socket.error`. В блокирующем режиме соответствующие вызовы блокируются до получения или отправки данных соответственно
`setsockopt(level, optname, value)`	Установка значения опции `optname` уровня `level` сокета в значение `value`. См. метод `getsockopt()` выше
`shutdown(how)`	Закрывает соединение полностью или частично. Если `how` равен 0, запрещается прием, 1 — передача, 2 — запрещаются и прием, и передача

В качестве примера можно привести клиент-серверную пару (листинги 9.61 и 9.62).

Листинг 9.61. Файл hello-client.py. Клиент

```
import socket

HOST = ""                 # удаленный компьютер (localhost)
PORT = 33333              # порт на удаленном компьютере
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST, PORT))
sock.send("ПАЛИНДРОМ")
result = sock.recv(1024)
sock.close()
print "Получено:", result
```

Листинг 9.62. Файл hello-server.py. Сервер

```
import socket, string

HOST = ""                 # localhost
PORT = 33333              # порт
srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.bind((HOST, PORT))
```

```
print "Слушаю порт 33333"
srv.listen(1)
sock, addr = srv.accept()
while 1:
  pal = sock.recv(1024)
  if not pal:
    break
  print "Получено от %s:%s:" % addr, pal
  lst = map(None, pal);
  lst.reverse();
  lap = string.join(lst, "")
  print "Отправлено %s:%s:" % addr, lap
  sock.send(lap)
sock.close()
```

Результат работы клиента:

```
Получено: МОРДНИЛАП
```

Результат работы сервера:

```
Слушаю порт 33333
Получено от 127.0.0.1:2484: ПАЛИНДРОМ
Отправлено 127.0.0.1:2484: МОРДНИЛАП
```

Модуль `socket` предоставляет множество других функций.

- `socket.getfqdn([hostname]) -> fqdname`

  Возвращает полное доменное имя хоста `hostname` или, если `hostname` не задан, дает значение от `gethostname()`.

  ```
  >>> import socket
  >>> socket.getfqdn("www")
  'www.onego.ru'
  ```

- `socket.gethostbyname(hostname) -> IP`

  Возвращает IP-адрес хоста `hostname`.

- `socket.gethostbyname_ex(hostname) -> (name, aliaslist, addrlist)`

  Возвращает имя `name`, список альтернативных имен `aliaslist` и список адресов `addrlist` хоста `hostname`.

  Например:

  ```
  >>> import socket
  >>> socket.gethostbyname_ex('www.whitehouse.gov')
  ```

```
('a1289.g.akamai.net',
['www.whitehouse.gov', 'www.whitehouse.gov.edgesuite.net'],
['64.242.116.8', '64.242.116.6'])
```

- `socket.gethostname() -> hostname`

  Возвращает первичное имя хоста, на котором исполняется эта команда. Параметр `hostname` необязательно является полным именем хоста.

- `socket.gethostbyaddr(ip_address) -> (hostname, aliaslist, addrlist)`

  Аналогично `socket.gethostbyname_ex`, но в качестве аргумента принимает IP-адрес.

- `socket.getprotobyname(proto_name) -> i`

  Возвращает номер протокола, соответствующий его символической записи (см. табл. 9.80). Может использоваться для задания третьего параметра при вызове функции `socket.socket()` для создания сокетов типа `SOCK_RAW`. В UNIX таблица с номерами протоколов приведена в файле /etc/protocols.

- `socket.getservbyname(servicename, proto_name) -> i`

  Возвращает номер порта для заданного Internet-сервиса и протокола (см. табл. 9.77). В UNIX-системах таблица соответствий сервисов и номеров портов обычно находится в файле /etc/services.

- `socket.htonl(i) -> i1`

  Меняет порядок байтов 32-разрядного целого `hi` на принятый в сети (то есть сначала старшие байты), если требуется (то есть если данный хост использует другой порядок байтов).

- `socket.ntohl(i) -> i1`

  Выполняет преобразование, обратное `htonl`.

- `socket.ntohs(i) -> i1`
  `socket.htons(i) -> i1`

  Функции, аналогичные `ntohl` и `htonl`, но для 16-разрядных целых чисел.

- `socket.inet_aton(addr) -> nnnn`

  Преобразует IP-адрес, записанный в строке `addr`, в четырехбайтную строку.

- `socket.inet_ntoa(nnnn) -> addr`

  Преобразует четырехбайтную строку с IP-адресом в "точечное" строковое представление.

В модуле `socket` определено большое количество различных констант (часть из них приведена выше в табл. 9.78, 9.79 и 9.80, а об остальных можно узнать из документации по UNIX). Имена констант начинаются на: `AF_`, `INADDR_`, `IPPORT_`, `IPPROTO_`, `IP_`, `MSG_`, `PACKET_`, `PF_`, `SOCK_`, `SOL_`, `SO_`, `TCP_`.

9.13. Поддержка Internet. Форматы данных

В этом разделе описаны модули для работы с форматами данных, обычно используемыми в сети Internet.

9.13.1. Модуль *quopri*

Модуль `quopri` предоставляет функции для преобразования данных из кодировки `quoted-printable` (QP) и обратно.

- `quopri.encode(input, output, quotetabs)`

  Читает данные из `input`, кодирует их и записывает в `output`. Флаг `quotetabs` — управляет кодированием табуляций. Объект `input` должен иметь метод `readline()`, а `output` — `write()`, т. е. `input` и `output` должны быть файлоподобными объектами.

- `quopri.decode(input, output)`

  Читает закодированные данные из файла `input`, раскодирует и записывает в файл `output`. Здесь `input` и `output` — файлоподобные объекты.

Листинг 9.63. Применение модуля `quopri` для кодирования текста

```
>>> from StringIO import StringIO
>>> import quopri
>>> text = """Это текст. \v\tThis is true: 2+2=4"""
>>> inp = StringIO(text)
>>> outp = StringIO()
>>> quopri.encode(inp, outp, 0)
>>> print outp.getvalue()
=FC=D4=CF =D4=C5=CB=D3=D4. =0B=09This is true: 2+2==4
```

В этом примере мы использовали модуль `StringIO`, чтобы не иметь дела с реальными файлами. Заметьте, что специальным образом кодируются только русские буквы, знак `{"="}` и управляющие символы (`"\v"` и `"\t"`).

9.13.2. Модуль *uu*

Модуль `uu` позволяет кодировать и декодировать данные методом `uuencode`, обычно применяющимся для кодирования вложений в электронных письмах. Модуль предоставляет исключение `uu.Error` и следующие функции.

- `uu.encode(in_path, out_path[, name, mode])`

  Кодирует файл с именем in_path в файл с именем out_path. Можно задать имя файла `name` и режим `mode` (по умолчанию — `0666`). Задание `"-"` вместо имени файла означает стандартный ввод (для in_path) или вывод (для out_path).

- `uu.decode(in_path[, out_path, mode])`

  Декодирует файл in_path в файл out_path. Если имя out_path и режим `mode` не заданы, они берутся из in_path (`uuencode` сохраняет имя файла и режим).

Следующий пример демонстрирует использование модуля `uu` для кодирования файла:

```
import uu
uu.encode("good.txt", "-")
```

Результат выполнения:

```
begin 664 good.txt
M66]U(&%R92!R:6=H="!T;R!T<GD@97AA;7!L97,@9G)O;2!T:&ES(&)O;VLA
M"O?9(-#2P=?)S-C.SR#0S]/4U=#!Q=3%+"#0TL_7Q=+1T2#0TLG-Q=+9(,G:
-(-S4S\H@R\[)Q\DA"@

end
```

9.13.3. Модуль *base64*

Модуль `base64` позволяет кодировать и декодировать строки и файлы с использованием кодировки base64, которая сейчас очень широко используется для кодирования электронных сообщений.

Модуль предоставляет две функции для кодирования.

- `base64.encode(input, output)`

  Кодирует файл input и записывает результат в файл output.

- `base64.encodestring(s) -> s1`

  Кодирует строку `s` в `s1`.

И две функции для декодирования.

- `base64.decode(input, output)`

  Декодирует файл input и записывает результат в файл output.

- `base64.decodestring(s) -> s1`

  Декодирует строку `s` в `s1`.

В следующем примере декодируется поле `subject` электронного письма:

```
import re, base64
subj = """Subject: =?koi8-r?B?UmU6IPfFwi3TwcrU?="""
def dcd(m): return base64.decodestring(m.group(3))
print re.sub('=\?(koi8-r)\?(b)\?(.*?)\?=', dcd, subj, re.I)
```

Результат выполнения:

```
Subject: Re: Веб-сайт
```

9.13.4. Модуль *binhex*

Модуль `binhex` позволяет работать с кодировкой binhex, применяемой на Macintosh. Следующие две функции помогут вам закодировать или декодировать файл с использованием binhex.

- `binhex.binhex(in_path, out_path)`

  Закодировать данные из файла in_path в файл out_path.

- `binhex.hexbin(in_path, out_path)`

  Декодировать данные из файла in_path в файл out_path.

```
from binhex import binhex
binhex("good.txt", "good.hqx")
```

В файле good.hqx будет следующее:

```
(This file must be converted with BinHex 4.0)

:#'G[Ef3ZG(Kd!$q3#!#3"@F!N!3R99P[G5"KFQ8JFQPRD(3JG'mJG(*j)'9iB@e
`E'9c)'CbEfdJG'KTFb"LEfpV)3Vhf5$3dX(AbFcBcXmJd-r6e0A3`FA8a5`Jd0,
2emA5dG%Jd0,*cFA5f5$*fL$Fe-r+)-[1bFI*)3UJJ`!!:
```

9.13.5. Модуль *binascii*

Модуль `binascii` реализован на C и предоставляет сразу несколько различных функций для перевода бинарных данных в ASCII и обратно, а также некоторые другие функции (табл. 9.82).

Таблица 9.82. *Основные функции модуля* `binascii`

Функция	Описание
`binascii.a2b_base64 (ascii) -> bin`	Декодирует строку в base64
`binascii.a2b_hex(ascii) -> bin`	Декодирует шестнадцатеричное представление данных

Таблица 9.82 (окончание)

Функция	Описание
`binascii.a2b_hqx(ascii) -> (bin, done)`	Декодирует hqx. Флаг `done` говорит о том, что встретился признак завершения закодированных данных (`{":"}`)
`binascii.a2b_uu(ascii) -> bin`	Декодирует данные, закодированные uuencode
`binascii.b2a_base64(bin) -> ascii`	Кодирует данные в base64
`binascii.b2a_hex(bin) -> ascii`	Возвращает шестнадцатеричное представление `bin`
`binascii.b2a_hqx(bin) -> ascii`	Кодирует данные в hqx
`binascii.b2a_uu(bin) -> ascii`	Кодирует строку в uuencode
`binascii.crc32(data [, oldcrc]) -> newcrc`	Подсчитывает контрольную сумму CRC для строки `data`. Если задан `oldcrc` (вычисленный для предыдущих данных), то будет вычислен итоговый CRC предыдущих данных и `data`

Следующий пример иллюстрирует подсчет контрольной суммы CRC32 текстового файла:

```
import binascii
f = open("file.dat", "rt")
crc = 0
for l in f.readlines():
  crc = binascii.crc32(l, crc)
print "CRC = %x" % crc
```

Результат выполнения:

```
CRC = 884863d2
```

9.13.6. Модуль *rfc822*

Модуль `rfc822` предоставляет класс `rfc822.Message` для работы с сообщениями в формате, определенном Internet-стандартом RFC-822 (например, таковыми являются сообщения электронной почты). Сообщения являются текстом и состоят из двух частей: *заголовочной части*, содержащей одно или несколько *полей (header fields)*, и *тела сообщения (body)*. Поля состоят из имени и значения. Поле с одним и тем же именем может повторяться в одном заголовке несколько раз.

Замечание

Описываемый здесь модуль не претендует на полное соответствие стандарту RFC-822. Например, поддерживаются не все схемы форматирования электронного адреса.

Класс `rfc822.Message` позволяет разбирать текст с сообщением, предоставлять доступ к полям заголовка и телу сообщения. Конструктор класса вызывается следующим образом:

```
rfc822.Message(file[, seekable]) -> mobj
```

где `file` — файлоподобный объект (объект, имеющий как минимум метод `readline()`) для ввода; `seekable` — флаг, который позволяет задавать, можно ли осуществлять произвольный доступ в файле `file` (если этот параметр не задан, конструктор попытается сам установить возможности объекта `file` и использовать их).

Класс может работать с любым файлоподобным объектом, поддерживающим метод `readline()`. Если объект поддерживает метод `seek()`, `rfc822.Message` будет использовать и его. Вводимые строки могут оканчиваться как на `"\n"`, так и на `"\r\n"`. Перед сохранением `"\r\n"` заменяются на `"\n"`. Имена полей нечувствительны к регистру букв, поэтому `"content-type"`, `"Content-Type"` и `"CONTENT-TYPE"` — одно и то же поле.

В табл. 9.83 рассмотрены методы экземпляров класса `rfc822.Message`. С объектом класса можно обращаться как со словарем, в котором ключами являются имена полей, а значениями — значения этих полей:

```
>>> import rfc822, StringIO
>>> msg = rfc822.Message(StringIO.StringIO(
...     """From: a\nTo: b1,\n  b1.1\nTo: b2\nSubject: c\n\nBody"""))
>>> print msg["from"]
a
>>> msg["from"] = "a1"
>>> print msg["to"]
b2
>>> del msg["to"]
>>> print msg["from"], msg.has_key("to")
a1 0
```

Следует заметить, что из двух полей `To` выбрано последнее в строке. Для получения информации обо всех одинаковых полях следует использовать команду `getallmatchingheaders()`.

Таблица 9.83. *Методы объектов класса* `rfc822.Message`

Метод	Описание
`getaddr(name) -> (comment, addr)`	Возвращает один адрес из поля `name` в виде кортежа, где `comment` — текстовый комментарий (обычно имя адресата), а `addr` — адрес
`getaddrlist(name) -> list`	Возвращает список кортежей с адресами из поля `name`
`getallmatchingheaders (name) -> list`	Возвращает все строки заголовочной части, относящиеся к полям с именем `name` или пустой список, если таких полей не нашлось
`getdate(name) -> date`	Возвращает дату, заданную в поле `name` в формате 9-элементного кортежа времени *(см. разд. 9.5.1)*
`getdate_tz(name) -> datetz`	То же, но десятым элементом кортежа идет значение смещения временной зоны источника сообщения от времени UTC. Например, для летнего московского времени это значение равно `14400`
`getfirstmatchingheader (name) -> list`	Возвращает список строк значения первого встреченного поля `name`
`getheader(name[, default]) -> value` или `get(...)`	Возвращает значение (последнего) поля `name` или `default`. По умолчанию `default` равен `None`
`getheaders(name) -> list`	Возвращает список значений для данного поля `name`
`getrawheader(name) -> s`	Аналогично `getfirstmatchingheader()` возвращает строку со значением поля `name` в том виде, в котором оно присутствует в заголовке сообщения или `None`, если поле не найдено
`has_key(name) -> 0, 1`	Возвращает `1`, если поле присутствует, и `0` в противном случае
`iscomment(line) -> 0, 1`	Определяет, является ли строка комментарием (и ее можно пропустить)
`isheader(line) -> name`	Определяет, является ди строка `line` правильной строкой с полем. Если является, возвращает название поля (`name`), иначе — `None`
`islast(line) -> 0, 1`	Определяет, является ли строка правильным окончанием заголовочной части сообщения
`items(), keys(), values()`	Методы для получения полей и их значений

Таблица 9.83 *(окончание)*

Метод	Описание
`readheaders()`	Читает строки заголовка и устанавливает атрибуты `status` и `headers`. Атрибут `status` равен пустой строке, если все нормально, а `headers` содержит необработанные строки прочитанного заголовка. (Этот метод вызывается в конструкторе класса `rfc822.Message`, поэтому атрибуты доступны сразу после создания экземпляра класса.)
`rewindbody()`	Устанавливает указатель позиции в файле сообщения на начало тела сообщения, если это возможно

Кроме того, у экземпляров класса доступны атрибуты: `headers()`, содержащий строки заголовочной части сообщения, и `fp`, в котором хранится исходный файловый объект.

9.13.7. Модуль *mimetools*

Модуль `mimetools` расширяет возможности модуля `rfc822` и предоставляет инструменты для работы с *MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения почты в Internet)*. Для этой цели модуль `mimetools` определяет класс `mimetools.Message` как подкласс класса `rfc822.Message`. О том, как создавать экземпляры класса `Message` *(см. разд. 9.13.6)*. А ниже будут описаны дополнительные возможности `mimetools.Message` по сравнению с `rfc822.Message` (табл. 9.84).

Таблица 9.84. *Дополнительные методы объектов класса* `mimetools.Message`

Метод	Описание
`getencoding()`	Возвращает кодировку, заданную в поле `content-transfer-encoding`
`getmaintype()`	Возвращает главный тип сообщения
`getparam(name) ->`	Возвращает значение первого параметра с именем `name` из результатов `getplist()`
`getplist() -> lst`	Возвращает список параметров поля `content-type` в виде списка строк `"ключ=значение"`, где параметр `ключ` приведен к нижнему регистру, а параметр `значение` остается как есть
`getsubtype()`	Возвращает подтип сообщения
`gettype()`	Возвращает тип сообщения в виде `"maintype/subtype"` (главный тип и подтип)

Модуль `mimetools` также определяет следующие функции.

- `mimetools.choose_boundary() -> s`

  Возвращает уникальную строку, которая с большой вероятностью подойдет в качестве строки-разделителя.

- `mimetools.decode(input, output, encoding)`

  Декодирует данные из файла input в файл output с использованием кодировки encoding: `"base64"`, `"uuencode"` или `"quoted-printable"`.

- `mimetools.encode(input, output, encoding)`

  Кодирует данные из файла input в файл output в кодировке encoding, см. выше.

- `mimetools.copyliteral(input, output)`

  Копирует из файла input в файл output построчно.

- `mimetools.copybinary(input, output)`

  Читает блоки из input до конца файла и записывает их в файл output. Размер блока `8192`.

Пример в листинге 9.65 демонстрирует применение `mimetools` для получения вложений из MIME-сообщений. Для проверки работы программы необходимо иметь MIME-сообщение, например, электронное письмо в чистом виде (для целей демонстрации лишние поля письма были вырезаны, а приложенный файл сокращен):

Листинг 9.64. Файл _message.txt. Файл с сообщением

```
>From 123@onego.ru  Thu Aug  2 15:24:34 2001
Subject: Hello
MIME-Version: 1.0
Content-Type: multipart/mixed;
    boundary="----=_NextPart_000_000C_01C11B02.BF9B6000"

This is a multi-part message in MIME format.

------=_NextPart_000_000C_01C11B02.BF9B6000
Content-Type: multipart/alternative;
       boundary="----=_NextPart_001_000D_01C11B02.BF9B6000"

------=_NextPart_001_000D_01C11B02.BF9B6000
Content-Type: text/plain; charset="koi8-r"
Content-Transfer-Encoding: quoted-printable
```

```
Hello!

------=_NextPart_001_000D_01C11B02.BF9B6000
Content-Type: text/html; charset="koi8-r"
Content-Transfer-Encoding: quoted-printable

<B>Hello!</B>

------=_NextPart_001_000D_01C11B02.BF9B6000--

------=_NextPart_000_000C_01C11B02.BF9B6000
Content-Type: application/msword; name="file.doc"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="file.doc"

0M8R4KGxGuEAAAAAAAAAAAAAAAAAAAAAPgADAP7/CQAGAAAAAAAAAAAAAAABAAAAMQAAAAA
AAAAAAAA

------=_NextPart_000_000C_01C11B02.BF9B6000—
```

Листинг 9.65. Файл mime_extract.py. Извлечение вложений

```
import mimetools, multifile, string

def extract_mime_parts(stream, name):
  """Записывает вложения в файлы на диске"""

  msg = mimetools.Message(stream)
  msgtype, params = msg.gettype(), msg.getplist()
  filename = string.join(name, ".")
  if msgtype[:10] == "multipart/":  # если часть составная:
    mf = multifile.MultiFile(stream)
    mf.push(msg.getparam("boundary"))
    cnt = 0
    while mf.next():
      cnt = cnt + 1
      extract_mime_parts(mf, name + (str(cnt),))
    mf.pop()
```

```
  else:  # иначе часть не содержит подчастей:
    outf = open(filename, "w")
    mimetools.decode(stream, outf, msg.getencoding())
    outf.close()
    print filename, "записан"

if __name__ == "__main__":
  extract_mime_parts(open("_message.txt", "r"), ("msg",))
```

Результат выполнения:

```
msg.1.1 записан
msg.1.2 записан
msg.2 записан
```

При выполнении программы все части MIME-сообщения были записаны на диск. В приведенном примере их оказалось три: две подчасти в первой части и файл с приложением во второй части. В некоторых сообщениях состав и расположение частей бывает более сложным.

9.13.8. Модуль *MimeWriter*

Модуль `MimeWriter` определяет класс `MimeWriter` для облегчения работы по созданию MIME-сообщений. Например, этот модуль можно использовать для формирования электронных писем с приложенными файлами. Класс `MimeWriter` имеет следующий конструктор:

```
MimeWriter.MimeWriter(fp) -> mwobj
```

где `fp` — файловый объект, в который можно записывать результат; `mwobj` — `MimeWriter`-объект.

Объекты класса `MimeWriter` имеют методы, представленные в табл. 9.85.

Таблица 9.85. *Методы объектов класса* `MimeWriter`

Метод	Описание
`addheader(key, value [, prefix]) ->`	Добавляет заголовок к MIME-сообщению. Аргумент `key` — название поля, `value` — его значение. Необязательный аргумент `prefix` указывает место, куда добавляется поле (`0` — в конец, `1` — в начало). По умолчанию `prefix` равен `0`
`flushheaders()`	Записывает (и забывает) накопленные к данному моменту поля

Таблица 9.85 *(окончание)*

Метод	Описание
`startbody(ctype [, plist[, prefix]]) -> fobj`	Начинает тело сообщения, возвращая позволяющий запись файловый объект. Параметр `ctype` указывает тип содержимого в виде `тип/подтип`, `plist` — содержит список кортежей с дополнительными параметрами к типу содержимого, `prefix` аналогичен одноименному параметру для `addheader()`, но по умолчанию равен `0`
`startmultipartbody (subtype[, boundary[, plist[, prefix]]]) -> fobj`	Начинает тело сообщения, в котором планируется несколько частей (то есть типа `multipart`). Параметр `subtype` указывает подтип (например, `mixed`, `alternative` или `signed`). Параметр `boundary` позволяет указать строку-разделитель, а `plist` и `prefix` действуют также, как и в `startbody()`. Части сообщения могут быть созданы с помощью метода `nextpart()`
`nextpart() -> mwobj`	Возвращает экземпляр `MimeWriter` для очередной части сообщения
`lastpart()`	Завершает последнюю часть сообщения

Пример в листинге 9.66 демонстрирует формирование сообщения электронной почты из двух частей: текстовой части и zip-файла. Из этого примера легко видеть последовательность шагов по формированию сообщения:

1. Создание объекта класса `MimeWriter`.
2. Задание полей с помощью метода `addheader()` этого объекта.
3. Получение файлового объекта для записи в тело сообщения с помощью методов `startbody()` и `startmultipartbody()`.
4. Запись в тело сообщения или выполнение метода `nextpart()`, если сообщение состоит из частей. После метода `nextpart()` для возвращенного объекта можно выполнить шаги, начиная со второго.

Листинг 9.66. Файл eletter.py. Формирование письма с вложенным файлом

```
import MimeWriter, sys, base64, zipfile
   # создаем сообщение (записываем его на стандартный вывод)
msg = MimeWriter.MimeWriter(sys.stdout)
msg.addheader("From", "book_reader@mail.ru")
msg.addheader("To", "rnd@onego.ru")
msg.addheader("Subject", "Letter and File")
msg.addheader("MIME-Version", "1.0")
```

```
body = msg.startmultipartbody("mixed")
body.write("This is a multi-part message in MIME format.\n")
   # начинаем первую (текстовую) часть
p1 = msg.nextpart()
p1.addheader("Content-Disposition", "inline")
p1f = p1.startbody("text/plain", [("charset", "koi8-r")])
p1f.write("Файл приложен к этому письму.")
   # начинаем вторую часть
p2 = msg.nextpart()
p2.addheader("Content-Transfer-Encoding", "base64")
p2.addheader("Content-Disposition", 'attachment; filename="file.zip"')
p2f = p2.startbody("application/x-zip-compressed",
                   [("name", "file.zip")])
   # готовим zip-файл
zf = zipfile.ZipFile("file.zip", "w", zipfile.ZIP_DEFLATED)
zf.write(sys.argv[0], "eletter.py")
zf.close()
   # кодируем файл в base64, записывая во вторую часть
base64.encode(open("file.zip", "r"), p2f)
   # закрываем сообщение
msg.lastpart()
```

9.13.9. Модуль *multifile*

Модуль `multifile` определяет класс `MultiFile` для работы с текстовыми файлами, состоящими из нескольких частей, разделенных специальными разделителями (таковыми, например, являются MIME-сообщения из нескольких частей).

Конструктор класса вызывается следующим образом:

```
multifile.MultiFile(fp[, seekable]) -> mfobj
```

где `fp` — файловый объект с методами `readline()`, `seek()` и `tell()` или просто с методом `readline()` (в этом случае аргумент `seekable` нужно установить равным `0`); `seekable` — флаг, указывающий, позволяет ли объект `fp` произвольный доступ; `mfobj` — экземпляр класса `MultiFile`.

Объект класса `MultiFile` каждую строчку входного файла `fp` рассматривает как строку с данными, *строку-разделитель* или строку с маркером завершения. Кроме того, допускаются вложения "файлов" друг в друга. (Здесь и далее при указании на файлоподобный объект для чтения части MIME-сообщения, слово файл будет браться в кавычки.) В табл. 9.86 приведено описание методов `MultiFile`. Следует заметить, что объекты класса

MultiFile являются файлоподобными объектами, т. е. в одном физическом файле (или сообщении) может быть вложено несколько других "файлов" (откуда и происходит название модуля: multifile).

Таблица 9.86. *Атрибуты объектов класса* MultiFile

Атрибут	Описание
end_marker(s) -> s1	Возвращает маркер завершения, соответствующий строке s
last	Истина, если последний конец "файла" произошел на маркере завершения
level	Уровень вложенности текущего "файла"
next() -> 0, 1	Переход к следующему "файлу". Возвращает значение "истина", если таковой "файл" имеется, иначе — значение "ложь"
pop()	Убирает строку, занесенную по push()
push(s)	Вносит строку, которая будет использоваться в строке-разделителе или маркере завершения "файла". Для чтения вложенных "файлов" может применяться несколько раз
read() -> s	Возвращает все оставшиеся данные из "файла". По сравнению с аналогичным методом настоящего файла, этот не принимает параметров
readline() -> s	Читает и возвращает строку "файла". Как и принято, при достижении конца "файла" возвращается пустая строка. При других видах ошибок (например, некорректных маркерах завершения и т. п.) могут быть возбуждены исключения multifile.Error
readlines() -> lst	Возвращает все оставшиеся строки "файла" в виде списка
section_divider(s) -> s1	Возвращает строку-разделитель, соответствующую строке s
seek(pos [, mode])	Устанавливает указатель на позицию pos в "файле". Параметр mode может принимать значения 0 (pos считается от начала), 1 (относительно текущей позиции) и 2 (относительно конца "файла")
tell() -> pos	Возвращает позицию указателя в "файле"

Следующий пример позволит лучше понять порядок применения функций модуля:

```
import multifile, StringIO
# файл с данными для примера:
df = StringIO.StringIO("""section 1
--b
```

```
section 2
--b
section 3
--b--
""")
mf = multifile.MultiFile(df)  # создаем мультифайл
mf.push("b")  # задаем строку-разделитель
while 1:      # читаем и выводим все "файлы"
  print `mf.read()`
  if not mf.next():  # пока они есть
    break
mf.pop()  # убираем строку разделитель
```

рабо тст .дулоьрь4ем

```
'section 1\012'
'section 2\012'
'section 3\012'
```

лорр 2рн раьд ун41рн ун4.р арь . *разд. 9.13.7.*

.13.10. Млы оь *mailbax*

Модуль `mailbox` предоставляет классы для работы с различными видами почтовых ящиков электронной почты (табл. 9.87).

Таблица 9.87. *Классы модуля* `mailbox`

Класс	Описание
`BabylMailbox (fp[, factory])`	Класс для доступа к почтовому ящику в формате Babyl (похож на MMDF)
`MHMailbox(dirname [, factory])`	Для доступа к MH-почтовому ящику. Параметр `dirname` содержит имя каталога, где хранятся сообщения
`Maildir(dirname [, factory])`	Для доступа к почтовому ящику в стиле Qmail. Параметр `dirname` указывает на каталог с набором ящиков. С помощью этого класса становится доступной новая (new) и текущая (current) почта
`MmdfMailbox (fp[, factory])`	Для доступа к почтовому ящику в стиле MMDF
`UnixMailbox (fp[, factory])`	Для доступа к классическому почтовому ящику UNIX, в котором сообщения содержатся в одном файле, разделенные строками вида "`From от время`". Параметр `fp` — файл с почтовым ящиком

Необязательный параметр `factory` позволяет указать класс сообщений. По умолчанию это — `rfc822.Message`. Может оказаться полезным указать в качестве `factory` класс `mimetools.Message`, как это сделано в примере ниже.

Работа с любым из этих классов очень проста. Объект любого из перечисленных классов имеет метод `next()`, который возвращает объект класса `rfc822.Message` (или класса, указанного в `factory`) для каждого сообщения в почтовом ящике. Возвращает `None`, когда сообщений нет.

Для работы примера в листинге 9.67 необходимо иметь файл `mbox` (в формате почтового ящика UNIX), а также файл `mime_extract.py` с листинга 9.65.

Листинг 9.67. Файл extract_mbox.py. Работа с почтовым ящиком

```
import mailbox, mime_extract, mimetools

mbf = open("mbox", "r")
mb = mailbox.PortableUnixMailbox(mbf, mimetools.Message)
msgcnt = 0
while 1:
  msg = mb.next()  # открываем очередное сообщение
  if not msg:
    break
  msgcnt = msgcnt + 1
  msgfrom = msg.get("from")
  print "Сообщение %5i от %s" % (msgcnt, msgfrom)
  msg.fp.seek(msg.startofheaders, 0)  # указатель на начало
  # извлекаем части MIME-сообщения:
  mime_extract.extract_mime_parts(msg.fp, ("msg%05i" % msgcnt,))
```

9.14. Python и языки разметки (SGML, XML, HTML)

Одной из сторон применения Python является его использование для обработки структурированных текстовых форматов. В этом разделе будут рассмотрены анализаторы трех различных языков разметки, имеющих поддержку на уровне стандартных библиотек Python. Следует заметить, что помимо стандартных модулей для тех же целей могут использоваться и модули сторонних производителей, которые можно отыскать в сети Internet.

9.14.1. Модуль *sgmllib*

Модуль `sgmllib` предоставляет класс `SGMLParser` для работы с подмножеством *SGML (Standard Generalized Markup Language, стандартный обобщенный язык разметки)*. Модуль ориентирован на обработку строгого подмножества SGML — языка HTML. Конструктор класса вызывается как `SGMLParser()`. Возвращаемый объект — SGML-анализатор — распознает следующие конструкции:

- открывающие и закрывающие теги в форме `<тег атрибут="значение" ...>` и `</тег>`
- числовая запись символа в виде `&#код;`
- запись указателя сущности (entity) в виде `&имя;`
- инструкция по обработке `<? ... >`
- комментарий `<!-- -->`

В табл. 9.88 приведены методы, которые определяет класс `sgmllib.SGMLParser`.

Таблица 9.88. *Методы объектов класса* `sgmllib.SGMLParser`

Метод	Описание
`reset()`	Сбрасывает объект, теряя все необработанные данные
`setnomoretags()`	Останавливает обработку тегов и читает весь последующий ввод буквально
`setliteral()`	Войти в режим буквального восприятия ввода (режим CDATA)
`feed(s)`	Подает строку `s` на вход анализатора. Цельные элементы обрабатываются, неполные откладываются в буфер. Может вызываться несколько раз, например, по мере поступления данных
`close()`	Вызывает обработку всех накопленных данных
`get_starttag_text()`	Возвращает текст последнего открывающего тега
`handle_starttag(tag, mtd, atts)`	Вызывается для манипуляции с открывающими тегами, для которых были определены методы `start_тег()` или `do_тег()`. Аргументу `tag` придается имя тега в нижнем регистре, `mtd` — метод для смысловой интерпретации открывающего тега, `atts` — список кортежей `(имя, значение)` для всех атрибутов тега. Имена атрибутов приведены к нижнему регистру. В базовом классе метод `handle_starttag()` просто вызывает `mtd(atts)`

Таблица 9.88 *(окончание)*

Метод	Описание
`handle_endtag (tag, mtd)`	Аналогичен `handle_starttag`, но вызывается для закрывающих тегов, для которых определен метод `end_тег()`. В базовом классе `sgmllib.SGMLParser` метод просто вызывает `mtd()`
`handle_data(data)`	Вызывается для обработки произвольных данных. В базовом классе этот метод ничего не делает. В производных классах ожидается его переопределение
`handle_charref(ref)`	Вызывается для обработки записей символов в виде числового кода: `&#код;`. Базовый класс поддерживает десятичные числовые коды в диапазоне 0–255, переводя их в ASCII и вызывая метод `handle_data()` с символом в качестве аргумента. Если `ref` некорректен или за пределами диапазона, вызывается метод `unknown_charref()` с аргументом `ref`
`handle_entityref (ref)`	Вызывается для обработки ссылок на сущности в виде `&имя;`. Имя используется в качестве ключа в словаре `entitydefs` (атрибут класса или объекта-анализатора), значениями которого являются расширения ссылок. Если расширение найдено, оно используется в качестве параметра в вызове `handle_data()`, а если нет — вызывается метод `unknown_entityref()` с аргументом `ref`. В базовом классе имеются расширения "`&`", "`'`", "`>`", "`<`" и '`"`' для сущностей `&`, `&apos`, `>`, `<` и `"` соответственно
`handle_comment (comment)`	Вызывается для обработки комментария. В качестве аргумента передается строка между признаками начала и конца комментария (без самих этих признаков). В базовом классе метод ничего не делает
`report_unbalanced (tag)`	Вызывается при нахождении закрывающего тега, для которого не было соответствующего открывающего
`unknown_starttag (tag, atts)`	Вызывается для обработки неизвестного открывающего тега
`unknown_endtag (tag)`	То же для неизвестного закрывающего тега
`unknown_charref (ref)`	Вызывается для обработки непонятного числового кода в форме `&#код;`
`unknown_entityref (ref)`	То же для ссылки на неизвестную сущность

Производные классы (а класс `sgmllib.SGMLParser` создан именно для наследования) могут переопределять не только приведенные в табл. 9.88, но и определять методы для обработки конкретных тегов.

- `start_тег(atts)`
- `do_тег(atts)`
- `end_тег()`

где `тег` — название тега (латинские буквы в нижнем регистре); `atts` — список кортежей `(имя, значение)`, соответствующий атрибутам тега.

Метод `start_тег()` определяет обработку открывающего тега. Метод `do_тег()` используется для обработки одиночных тегов (не имеющих закрывающего тега). Метод `end_тег()` вызывается для встреченного закрывающего тега.

Пример в листинге 9.68 показывает работу SGML-анализатора в действии.

Листинг 9.68. Работа SGML-анализатора

```
import sgmllib, urllib

example = """<FORM NAME="form2" ACTION="script.cgi" METHOD="GET">
<BR><INPUT TYPE="submit" NAME="B2" VALUE="Очистить"></FORM>
<FORM NAME="form1" ACTION="script.cgi" METHOD="GET">
Имя: <INPUT TYPE="text" SIZE="50" NAME="name" VALUE="1" VALUE="2">
<BR>Текст:<TEXTAREA ROWS="2" COLS="20" NAME="text"></TEXTAREA>
<BR><INPUT TYPE="submit" NAME="B1" VALUE="Отправить"></FORM>"""

class FormParser(sgmllib.SGMLParser):
  """Класс анализаторов форм в HTML-документе.
  Выдает значение формы в виде строки запроса."""
  def __init__(self, which_form=None):
    sgmllib.SGMLParser.__init__(self)
    self._record = 0  # признак активности записи полей формы
    self._which_form = which_form  # имя выбираемой формы
  def __str__(self):
    return urllib.urlencode(self._form)
  def start_form(self, atts):
    """Начинаем обрабатывать форму"""
    attdict = self._attdict(atts)
    # если имя формы подходящее, начинаем запись полей формы
```

```
    if self._which_form is None or self._which_form == attdict["name"]:
      self._record = 1
      self._form = {}
  def end_form(self):
    """Завершаем обработку формы"""
    self._record = 0
  def _attdict(self, atts):
    """Составление словаря атрибутов"""
    attdict = {}
    for attname, attval in atts:
      attdict[attname] = attval
    return attdict
  def start_input(self, atts):
    """Обработчик тега формы"""
    if self._record:
      attdict = self._attdict(atts)
      if attdict.has_key("name"):
        name = attdict["name"]
        if attdict.has_key("value"):
          self._form[name] = attdict["value"]
        else:
          self._form[name] = ""
  # обрабатываются аналогично:
  start_select = start_textarea = start_input

f = FormParser();        f.feed(example); print str(f)
f = FormParser("form2"); f.feed(example); print str(f)
```

Результат выполнения:

```
text=&name=2&B1=%ef%d4%d0%d2%c1%d7%c9%d4%d8
B2=%ef%de%c9%d3%d4%c9%d4%d8
```

В приведенном примере анализатор "реагирует" только на теги `FORM`, `INPUT`, `TEXTAREA` и `SELECT`, которые используются в формах. Причем, для тега `FORM` обрабатывается как открывающий, так и закрывающий теги, а у всех остальных обрабатывается только открывающий. Обработчики вызываются в порядке следования тегов в тексте `example`, поэтому мы имеем возможность выбрать форму по имени.

9.14.2. Модуль *htmllib*

Описанный ранее модуль `sgmllib` является базой для модуля `htmllib`. Модуль определяет класс `htmllib.HTMLParser`, он наследуется от `sgmllib.SGMLParser`, с конструктором

```
HTMLParser(formatter) -> htmlparser
```

Здесь в качестве аргумента конструктора класса передается объект, который управляет форматированием. В этом качестве годятся, например, экземпляры класса `formatter.AbstractFormatter` из модуля `formatter` (который в данной книге не рассматривается). Из того же модуля можно взять класс `formatter.DumbWriter` для вывода текста.

Листинг 9.69. Файл html2txt.py. Пример конвертора из HTML в текст

```
import htmllib, formatter, urllib

# читаем URL из Internet:
url = "http://www.gospelcom.net/cqod/"
ex = urllib.urlopen(url).read()

# используем предлагаемые модулем formatter средства
# для записи
w = formatter.DumbWriter()
f = formatter.AbstractFormatter(w)
h = htmllib.HTMLParser(f)
h.feed(ex)
```

В табл. 9.89 перечислены дополнительные (по сравнению с классом `sgmllib.SGMLParser`) атрибуты класса `htmllib.HTMLParser`.

Таблица 9.89. *Методы объектов класса* `htmllib.HTMLParser`

Атрибут	Описание
`formatter`	Объект форматирующего класса (например, из модуля `formatter`)
`Nofill`	Флаг, при истинном значении которого повторные пробельные символы не будут сжиматься до одного. По умолчанию значение флага — "ложь"
`anchor_bgn (href, name, type)`	Вызывается в начале области, отмеченной якорем ссылки (тег `<A>`). Параметры `href` и `name` получают значения соответствующих атрибутов

Таблица 9.89 *(окончание)*

Атрибут	Описание
`anchor_end()`	Вызывается в конце обозначенной якорем области (тег `</A>`)
`handle_image (source, alt [, ismap[, align [, width[, height]]]])`	Вызывается для тега `<IMG>`, из атрибутов которого получают значения аргументы метода
`save_bgn()`	Начинает сохранять символьные данные в буфере (вместо их передачи `formatter`-объекту)
`save_end()`	Завершает сохранение данных в буфере. Данные обрабатываются в соответствии с методом `nofill`. Вызовы `save_bgn()`/`save_end()` не могут быть вложенными

В версиях 1.5, 1.6 для организации собственной обработки HTML-документа лучше всего, наверное, использовать модуль `sgmllib`, либо приспособить интерфейсы классов `formatter.NullFormatter` и `formatter.NullWriter` для своих нужд. В табл. 9.90 и 9.91 приводятся краткие описания интерфейсов этих классов.

Конструктор класса `formatter.NullFormatter` имеет вид

```
NullFormatter([writer])
```

где `writer` — объект, который будет записывать данные, поставляемые объектом класса `NullFormatter` (или производного от него класса)

```
def __init__(self, writer=None):
if not writer:
writer = NullWriter()
self.writer = writer
```

Таблица 9.90. *Интерфейс класса* `formatter.NullFormatter`

Метод	Описание
`end_paragraph (blankline)`	Закрывает абзац и записывает не менее `blankline` пустых строк перед следующим абзацем
`add_line_break()`	Добавляет разрыв строки (не разрывая логический абзац)
`add_hor_rule(...)`	Вставляет горизонтальную линию (не разрывая логический абзац). Аргументы передаются методу `send_line_break()` при работе с `writer`
`add_label_data (format, counter)`	Добавляет пометку слева от текущей левой границы. Используется для создания нумерованных или маркированных списков и т. п. Параметр `format` является форматом для целого числа `counter`. Например, формат `"i"` указывает на то, что должны выводиться строчные римские цифры: `i`, `ii`, `iii`, ...

Таблица 9.90 *(окончание)*

Метод	Описание
`add_flowing_data (data)`	Добавляет символьные данные, в которых повторные пробельные символы сжимаются до одного пробела
`add_literal_data (data)`	То же, но не изменяет пробельных символов
`flush_softspace()`	Отправляет объекту `writer` все накопленные от предыдущего вызова `add_flowing_data()` пробельные символы
`push_alignment (align)`	Кладет новое значение выравнивания `align` на стек выравниваний. Значение `formatter.AS_IS` служит для указания отсутствия изменений выравнивания
`pop_alignment()`	Снимает со стека значение выравнивания
`push_font ((size, italic, bold, mono))`	Изменяет значения характеристик шрифта (размер, курсив, жирность, моноширинность). Константа `formatter.AS_IS` может использоваться для указания того, что характеристику нужно оставить как есть
`pop_font()`	Восстанавливает предыдущее характеристики шрифта
`push_margin (margin)`	Увеличивает величину отступов слева на единицу
`pop_margin()`	Восстанавливает предыдущее значение отступа слева
`set_spacing (spacing)`	Устанавливает стиль разбивки
`push_style(style1, style2, ...)`	Кладет несколько спецификаций стилей на стек стилей. Этот стек передается методу `writer.new_styles()`
`pop_style([n])`	Убирает `n` (по умолчанию `n` равно 1) с вершины стека стилей
`assert_line_data ([flag])`	Информирует объект о том, что к текущему абзацу были добавлены данные путем прямого обращения к `writer`. Флаг `flag`, установленный в 1 (по умолчанию), говорит о том, что разрыва строки не было сделано

Таблица 9.91. *Интерфейс класса `formatter.NullWriter`*

Метод	Описание
`flush()`	Сбрасывает буфер вывода
`new_alignment (align)`	Устанавливает выравнивание `align`
`new_font(font)`	Устанавливает стиль шрифта `font`

Таблица 9.91 *(окончание)*

Метод	Описание
`new_margin(margin, level)`	Устанавливает левый отступ на значение `level` с логической пометкой `margin`. Значение `margin` не должно быть ложным для ненулевых значений `level`. Например, `level` может означать уровень вложенности списка, а `margin` — вид списка
`new_spacing (spacing)`	Устанавливает новый стиль разбивки
`new_styles(styles)`	Устанавливает дополнительные стили `styles` (кортеж произвольных значений, интерпретация которых зависит от приложения)
`send_paragraph (blankline)`	Отделяет абзац не менее чем `blankline` строками
`send_line_break()`	Разрывает строку
`send_hor_rule(...)`	Показывает горизонтальную линию. Параметры зависят от приложения
`send_label_data (data)`	Данные для пометки слева от границы абзаца. Например, номер пункта списка. Зависит от приложения
`send_flowing_data (data)`	Данные текстового потока, который может быть отформатирован. Подразумевается, что повторные пробелы сжаты до одного
`send_literal_data (data)`	Буквальные данные

Описанные выше классы для объектов `writer` и `formatter` могут использоваться и сами по себе. В любом случае, они предоставляют хорошую основу для несложных приложений, в которых требуется преобразовывать одну разметку в другую, т. к. достаточно следовать описанным интерфейсам и реализовать соответствующие методы. Для более сложных работ могут потребоваться более мощные средства, например, описанный в следующих разделах XML, часто даже XSLT (входит в состав набора пакетов 4Suite от Fourthought, Inc., который можно взять на Web-сайте **www.4Suite.org/download.HTML#4Suite** или на прилагаемом к этой книге компакт-диске).

В стандартной поставке Python также имеется модуль `htmlentitydefs`, в котором определен словарь `entitydefs`, устанавливающий соответствие между именем сущности и ее значением в HTML (в предположении, что используется кодировка latin-1). Например:

```
>>> import htmlentitydefs
>>> print htmlentitydefs.entitydefs["gt"]    # &lt;
```

```
<
>>> print htmlentitydefs.entitydefs["Ouml"]  # &Ouml;
Ж
```

9.14.3. Пакет *xml*

XML (Extensible Markup Language, расширяемый язык разметки) — одно из новейших достижений в области информационных технологий, на которое возлагаются большие надежды по объединению разрозненных приложений в единый ансамбль, пользующийся гибким расширяемым форматом для обмена структурированными данными.

Язык XML так же, как и HTML, является потомком SGML *(см. разд. 9.14.1)*, но, в отличие от HTML, не является строгим подмножеством SGML и не ограничен разметкой документов. Это — видоизмененная и упрощенная версия (диалект) SGML. Также, в отличие от HTML, в XML можно создавать свои собственные наборы тегов, ориентированные на конкретную предметную область. На XML можно записывать базы данных и знаний, протоколы взаимодействия между объектами, описания ресурсов и многое другое.

Тем, кто только начинает знакомиться с XML, не всегда понятно, зачем он нужен, и еще менее понятно — как же с ним работать. Не вдаваясь в детали, можно сказать, что XML вносит в данные контекстную информацию, наличие которой позволяет программам гораздо легче "понимать" смысл данных, а приложениям от разных производителей — взаимодействовать. Сравните следующие примеры:

```
data = ('20002002', 'USD', 2640)
data = {'дата': '20002002', 'валюта': 'USD', 'стоимость': 2640}
```

В первом случае информация о смысле хранимых в кортеже величин начисто отсутствует, поэтому их интерпретация полностью зависит от некоторых внешних спецификаций, которые ставят соответствие между номером позиции в кортеже и смыслом величины, стоящей в этой позиции. Во втором случае присутствует контекстная информация, а потому формат стал более гибким: можно добавлять новые элементы без риска спутать величины с различным смыслом. Однако и здесь есть некоторые неясности, например, в каких единицах измеряется стоимость валюты. Тем самым, мы приходим к более сложной структуре данных, которая легко записывается на XML. Например, программа в листинге 9.70 выдает следующий результат (для наглядности он был немного приукрашен):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<currency_data>
  <record date="20002002" currency="USD">
    <cost currency="RUR" factor="100">2640</cost>
```

```
  </record>
</currency_data>
```

Листинг 9.70. Файл xml_gen.py. Генератор XML

```
from xml.sax.saxutils import XMLGenerator
g = XMLGenerator()
g.startDocument()
g.startElement("currency_data", {})
g.startElement("record", {"date": "20002002", "currency": "USD"})
g.startElement("cost", {"currency": "RUR", "factor": "100"})
g.characters("2640")
g.endElement("cost")
g.endElement("record")
g.endElement("currency_data")
g.endDocument()
```

Конечно, приведенную структуру данных можно записать и в другом формате, но XML предлагает единый общепринятый стандарт и наличие стандартных средств обработки, так что сложно удержаться от его использования. Не нужно обращать внимание на то, что объем данных возрос в несколько раз по сравнению с первым примером: хранение и передача данных сейчас обходятся дешево. Куда дороже может обойтись отсутствие контекстной информации в нужный момент в нужном месте.

Говоря об XML, надо иметь в виду, что XML-документы бывают формально-правильными (well-formed) и состоятельными (valid). Приведенный выше пример является примером формально-правильного XML. Состоятельные XML-документы — это формально-правильные XML-документы, имеющие при себе *объявление типа документа (DTD, Document Type Definition)*, который строго задает структуру (грамматику) документа. Разумеется, документ должен удовлетворять этой структуре. Соответственно, анализаторы XML делятся на непроверяющие (non-validating) и проверяющие (validating) в зависимости от того, проверяют ли они соответствие документа объявленному типу (состоятельность).

Описанные далее модули помогут эффективно работать с XML, а приведенные примеры помогут понять основы работы с XML на Python. В версиях Python 2.0 и старше не рекомендуется использовать пкеты `sgmllib`, `htmllib` и `xmllib`. Вместо двух первых пакетов рекомендуется использовать `HTMLParser`, позволяющий работать с содержанием на XHTML. Пакет `xml` появился в Python 2.0 и пришел на смену модулю `xmllib`. Части этого пакета мы и рассмотрим далее.

Пакет *xml.sax*

Пакет `xml.sax` содержит несколько модулей, реализующих *SAX (Simple API for XML, простой программный интерфейс для XML).*

Работа SAX заключается в чтении источников данных (input source) XML-анализаторами (XML-reader) и порождении последовательности событий (events), которые обрабатываются объектами-обработчиками (handlers). Обычно SAX направлен на построение объектной модели, и события служат теми командами, которые приводят к построению этой модели. Например, объекты-обработчики могут заполнять должным образом внутренние структуры данных программы. Конечно, ничто не мешает обработчикам служить посредниками между XML-описанием некоторого процесса и его реализацией некоторым устройством или устройствами. В этом случае события переводятся в команды устройства.

В листингах 9.71, 9.72 и 9.73 с помощью SAX вычисляется заданное на XML арифметическое выражение.

Источником данных служит объект класса `InputSource` из модуля `xml.sax.xmlreader`. В качестве ввода из последовательности байтов берется пример из файла arith.xml.

В примере использован готовый XML-анализатор (в стандартном Python поставляется `expat`), экземпляр которого мы получаем вызовом функции `xml.sax.make_parser()`.

К этому анализатору мы присоединяем наши обработчики событий, которые мы определили в модуле `EvalExpr`. На самом деле, мы изменили только `ContentHandler`, взяв все остальные готовыми. Этому классу-обработчику XML-анализатор направляет все события, которые возникают при прочтении источника XML. Нас интересуют только открывающие и закрывающие теги, а также данные, которые находятся между ними, что и нашло свое отражение в реализации методов класса `EvalExpr.ContentHandler`.

Выражение записывается с помощью тегов `<expression>`, `<operation>`, `<operand>`, `<type>`. Вычисления производятся в порядке, заданном вложением тегов друг в друга, поэтому отложенные операции заносятся в стек `self.stack`

Листинг 9.71. Файл EvalExpr.py. Вычисление арифметического выражения

```
import string
from xml.sax import handler


class Operation:
  """Класс для хранения операции с операндами"""
  def __init__(self): self._operands = []
```

```
  def set_type(self, type): self._type = type
  def set_operand(self, operand): self._operands.append(operand)
  def eval(self): return eval(str(self))
  def __str__(self):
    """Представление операции в виде строки"""
    if len(self._operands) == 1:
      return "(%s(%s))" % (self._operands, self._type)
    else:
      return "(%s)" % string.join(
          map(str, self._operands), str(self._type))

class ContentHandler(handler.ContentHandler):
  """Класс, методы которого вызываются для обработки событий"""
  def __init__(self):
    self.cur_data = ""   # накопитель данных
    self.stack = []      # стек для хранения отложенных операций
    self.value = None    # текущее значение

  def startElement(self, name, attrs):
    """Встретился открывающий тег name"""
    if name == "operation": self.stack.append(Operation())
    elif name == "operand": self.value = None
    self.cur_data = ""

  def endElement(self, name):
    """Встретился закрывающий тег name"""
    if name == "expression": self.value = str(self.value.eval())
    elif name == "operation": self.value = self.stack.pop()
    elif name == "type": self.stack[-1].set_type(self.cur_data.strip())
    elif name == "operand":
      self.value = self.value or self.cur_data.strip()
      self.stack[-1].set_operand(self.value)
    self.cur_data = ""

  def characters(self, data):
    """Встретились "межтеговые" данные"""
    self.cur_data += data

# Эти классы тоже можно было бы переопределить, но
# мы оставим их как есть:
```

```
# обработчик ошибок, ссылок на сущности и событий DTD-части
class ErrorHandler(handler.ErrorHandler): pass
class EntityResolver(handler.EntityResolver): pass
class DTDHandler(handler.DTDHandler): pass
```

Листинг 9.72. Файл 2plus2.py. Пример использования модуля `EvalExpr`

```
import xml.sax
from xml.sax.xmlreader import InputSource

# Готовим обработчики событий:

from EvalExpr import *
content_handler = ContentHandler()
error_handler = ErrorHandler()
entity_resolver = EntityResolver()
dtd_handler = DTDHandler()

# Готовим источник данных:

input = open("arith.xml", "r")
input_source = InputSource()
input_source.setByteStream(input)

# Готовим XML-анализатор:

# Используем готовый анализатор:
xml_reader = xml.sax.make_parser()

# Регистрируем обработчики:
xml_reader.setContentHandler(content_handler)
xml_reader.setErrorHandler(error_handler)
xml_reader.setEntityResolver(entity_resolver)
xml_reader.setDTDHandler(dtd_handler)

# Анализируем источник XML:
xml_reader.parse(input_source)

# Выводим результат вычислений:
print content_handler.value
```

Листинг 9.73. Файл arith.xml. Файл с тестовыми данными

```
<?xml version="1.0" encoding="iso-8859-1"?>
<expression><operation>
  <type>+</type>
  <operand>2</operand>
  <operand>
    <expression><operation>
      <type>*</type>
      <operand>3</operand>
      <operand>4</operand>
    </operation></expression>
  </operand>
</operation></expression>
```

Теперь, когда идея SAX стала более понятной, можно поближе познакомиться с модулями, которые предлагает пакет `xml.sax`.

Модуль *xml.sax.handle*. В этом модуле определены четыре класса объектов-обработчиков и связанные с ними свойства. В табл. 9.92 описаны классы обработчиков, а в табл. 9.93 перечислены их свойства и особенности.

Таблица 9.92. *Классы объектов-обработчиков для SAX*

Класс	Описание	Таблица
`ContentHandler`	Обработчик содержимого. События поступают на интерфейс объектов этого класса в порядке следования данных в документе	9.94
`DTDHandler`	Обработчик DTD-событий, получающий сообщения о событиях при обработке DTD-части документа	9.95
`EntityResolver`	Базовый интерфейс для разбора сущностей. Анализатор будет использовать его для разрешения внешних сущностей	9.96
`ErrorHandler`	Обработчик ошибок, который позволяет анализатору сообщать об ошибках и предупреждениях. Например, обработка ошибок в базовом классе `xml.sax.handler.ErrorHandler` заключается в возбуждении исключений	

Константы табл. 9.93 тоже определены в модуле `xml.sax.handler` и могут использоваться с методами `setFeature()`, `getFeature()`, `f:setProperty`, `getProperty()` объектов класса `XMLReader`.

Таблица 9.93. *Константы для свойств и особенностей SAX*

Свойство	Значение
`feature_namespaces`	Выполнять обработку пространств имен (namespaces). Предполагает `feature_namespace_prefixes`
`feature_namespace_prefixes`	Возвращать исходные имена тегов и атрибутов, использованные при объявлении пространств имен
`feature_string_interning`	Применять встроенную функцию `intern()` для всех символических имен
`feature_validation`	Сообщать о несостоятельностях документа (предполагает `feature_external_ges` и `feature_external_pes`)
`feature_external_ges`	Включать все внешние обобщенные сущности (general entities)
`feature_external_pes`	Включать все параметрические сущности (parameter entities)
`property_lexical_handler`	Необязательный обработчик лексических событий (например, комментариев). Не поддерживается по умолчанию
`property_declaration_handler`	Необязательный обработчик DTD-части документа (для событий, которые не обрабатывает `DTDHandler`). Не поддерживается по умолчанию
`property_dom_node`	Указывает текущий узел DOM или корень документа. Не поддерживается по умолчанию
`property_xml_string`	Строка с данными, являющаяся источником данного события

Методы в табл. 9.94—9.96 вызываются XML-анализаторами при разборе XML-документов:

Таблица 9.94. *Методы объектов класса* `ContentHandler`

Метод	Событие и его описание
`characters(data)`	Символьное содержимое
`endDocument()`	Конец документа
`endElement(name)`	Закрывающий тег
`endElementNS(name, qname)`	Закрывающий тег в режиме обработки пространств имен. Параметры те же, что и у `startElementNS()`
`endPrefixMapping(prefix)`	Окончание области соответствия префикса `prefix` своему пространству имен
`ignorableWhitespace()`	Пробельные символы, которые можно игнорировать

Таблица 9.94 *(окончание)*

Метод	Событие и его описание
`processingInstruction (target, data)`	Инструкция по обработке
`setDocumentLocator (locator)`	Вызывается анализатором для передачи указателя (locator) документа перед вызовом всех других методов
`skippedEntity(name)`	Пропущенная сущность
`startDocument()`	Начало документа
`startElement(name, attrs)`	Открывающий тег `name` и его атрибуты — словарь `attrs`
`startElementNS(name, qname, attrs)`	Открывающий тег в режиме обработки пространства имен, аналогично `startElement()`. Параметр `name` содержит кортеж `(URI, local)`. Параметр `qname` — сырое имя, использованное в XML-документе, а `attrs` — объект класса `attributesNS`, содержащий атрибуты
`startPrefixMapping (prefix, URI)`	Начало области соответствия префикса `prefix` и URI пространства имен. Различные пространства необязательно вложены друг в друга, как это происходит с тегами. Гарантируется только, что начало конкретного пространства имен идет перед его окончанием

Если в документе присутствует DTD, могут возникнуть события, обрабатываемые методами, приведенными в табл. 9.95.

Таблица 9.95. *Методы объектов класса* `DTDHandler`

Метод	Событие и его описание
`notationDecl(name, publicId, systemId)`	Определение нотации `name` с общим идентификатором `publicId` и системным идентификатором `systemId`
`unparsedEntityDecl (name, publicId, systemId, ndata)`	Определение неразобранной сущности. Здесь `ndata` — данные неразобранной сущности

Для разрешения сущностей служат методы, представленные в табл. 9.96.

Таблица 9.96. *Методы объектов класса* `ErrorHandler`

Метод	Событие и его описание
`resolveEntity(public Id, systemId)`	Разрешает системный идентификатор сущности и возвращает либо `systemId` (по умолчанию) как строку для чтения, либо объект `InputSource`

Возникающие при разборе XML-документа ошибки делятся на три категории: необратимые (фатальные), поправимые и предупреждения (табл. 9.97). Продолжение анализа зависит от того, как поступит обработчик ошибок. Если он возбудит исключение, анализ прекратится. По умолчанию все три категории ошибок вызывают исключение.

Таблица 9.97. *Ошибки, возникающие при разборе XML-документа*

Ошибка	Описание
`fatalError (exc)`	Фатальная ошибка, ведущая к прекращению анализа документа
`error(exc)`	Поправимая ошибка. Дальнейший анализ документа может привести к новым ошибкам. Дальнейшее получение событий на основании анализируемого XML-документа не гарантируется
`warning(exc)`	Предупреждение, после которого анализ может быть продолжен

Модуль *xml.sax.saxutils*. Этот модуль предлагает функции и классы, часто используемые при работе с SAX. Классы могут как использоваться на прямую, так и служить базой для других классов, более приспособленных к нуждам пользователя.

- `saxutils.escape(s[, entities]) -> s1`

  Функция для экранирования особых символов строки `s` с целью безопасного включения в XML-текст. Если параметр `entities` не задан, символы "`&`", "`<`" и "`>`" заменяются на ссылки на соответствующие сущности. Параметр `entities` содержит словарь, с использованием которого происходят замены в строке `s`. Следующий пример показывает буквальное включение HTML-кодов в XML:

  ```
  >>> from xml.sax.saxutils import escape
  >>> print escape('<A href="1">&entity;</A>', {'"': "&quot;"})
  &lt;A href=&quot;1&quot;&gt;&amp;entity;&lt;/A&gt;
  ```

  `saxutils.XMLGenerator([out[, encoding]]) -> handler`

  где `out` — файлоподобный объект для вывода (по умолчанию — `sys.stdout`); `encoding` — кодировка (по умолчанию iso-8859-1).

  Этот класс мы уже использовали в примере для построения XML-кода. Если объект этого класса задать в качестве обработчика `ContentHandler`, мы получим исходный XML-документ.

- `saxutils.prepare_input_source(source[, base]) -> input_source`

  Функция принимает источник данных `source` (строка, файлоподобный объект, объект `InputSource`) и базовый URL, а возвращает готовый к ис-

пользованию `InputSource`-объект. Функция может быть использована в анализаторах XML для адаптации типа источника данных.

Модуль *xml.sax.xmlreader*. Модуль `xml.sax.xmlreader` содержит несколько классов, имеющих отношение к источнику данных, а также к интерфейсу `XMLreader`, который должен реализовать XML-анализатор при работе по SAX.

Класс `xml.sax.InputSource` несколько больше, чем аналог файлоподобного объекта: он включает в себя мета-информацию вроде общего и системного идентификаторов и поток байтов или символов. Класс `InputSource` используется как на входе анализатора, так и возвращается методом `resolveEntity()` класса `EntityResolver`. Методы класса `xml.sax.InputSource` даны в табл. 9.98.

Класс `XMLReader` служит базовым классом для XML-анализаторов. Его разновидность — `IncrementalParser` — пригодится в случаях, когда документ недоступен сразу во всей полноте. В табл. 9.99 содержатся методы экземпляров этих классов.

Модуль также определяет класс `Locator` для хранения указателя во время анализа XML-документа. Его методы приведены в табл. 9.100.

Таблица 9.98. *Методы объектов класса* `xml.sax.InputSource`

Метод	Описание
`setPublicId(id)`	Задает общий идентификатор источника данных
`getPublicId()`	Возвращает общий идентификатор источника данных
`setSystemId(id)`	Задает системный идентификатор
`getSystemId()`	Возвращает системный идентификатор
`setEncoding(encoding)`	Задает кодировку источника данных. Кодировка игнорируется, если задан поток символов
`getEncoding()`	Возвращает кодировку
`setByteStream(bytefile)`	Задает поток байтов, связанный с источником данных, — файлоподобный объект. Игнорируется, если задан поток символов
`getByteStream()`	Возвращает объект, связанный с потоком байтов или `None`, если таковой не был задан
`setCharacterStream (charfile)`	Задает поток Unicode-символов (файлоподобный объект)
`getCharacterStream()`	Возвращает файлоподобный объект, связанный с потоком Unicode-символов

Два других класса, которые мы не будем рассматривать в деталях: `AttributesImpl` и `AttributesNSImpl`. Экземпляры этих классов представляют атрибуты тегов XML (`NS` означает namespace и говорит о том, что атрибуты учитывают пространство имен).

Таблица 9.99. *Методы объектов класса* `XMLReader` *и* `IncrementalParser`

Метод	Описание
`parse(source)`	Обрабатывает источник XML-данных `source`, вызывая соответствующие обработчики событий. Источник данных может быть именем файла, URL, файлоподобным объектом или объектом `InputSource`
`getContentHandler()`	Возвращает текущий обработчик содержимого
`setContentHandler(handler)`	Устанавливает обработчик содержимого
`getDTDHandler()`	Возвращает обработчик DTD
`setDTDHandler(handler)`	Устанавливает обработчик DTD
`getEntityResolver()`	Возвращает объект для разрешения сущностей
`setEntityResolver(handler)`	Устанавливает объект для разрешения сущностей
`getErrorHandler()`	Возвращает объект для обработки ошибок
`setErrorHandler(handler)`	Устанавливает объект для обработки ошибок
`setLocale(locale)`	Устанавливает locale (культурную среду) для ошибок и предупреждений. Если анализатор не поддерживает locale, он должен вызывать исключение
`getFeature(featurename)`	Возвращает особенность `feature`
`setFeature(featurename, value)`	Устанавливает особенность `feature` в значение `value`
`getProperty(propertyname)`	Возвращает свойство `propertyname`
`setProperty(propertyname, value)`	Устанавливает свойство `propertyname` в значение `value`
`feed(data)`	Вводит порцию данных (только для `IncrementalParser`)
`close()`	Подразумевает конец документа (только для `IncrementalParser`)
`reset()`	Переустановка анализатора в исходное состояние (только для `IncrementalParser`)

Примечание

Методы инкрементного анализатора можно использовать только между вызовами `parse()`.

Таблица 9.100. *Методы объектов класса* `Locator`

Метод	Описание
`getColumnNumber()`	Возвращает номер колонки конца участка документа, соответствующего текущему событию
`getLineNumber()`	Возвращает номер строки, соответствующий текущему событию
`getPublicId()`	Общий идентификатор текущего события
`getSystemId()`	Системный идентификатор текущего события

Попробуйте добавить следующие строки к классу `ContentHandler` из листинга 9.71 — вы получите точные координаты окончания каждого кусочка данных.

```
# в класс ContentHandler:
  def setDocumentLocator(self, locator):
    self._loc = locator

  def characters(self, data):
    """Встретились "межтеговые" данные"""
    print self._loc.getColumnNumber(), self._loc.getLineNumber(),
    print self._loc.getPublicId(), self._loc.getSystemId()

    self.cur_data += data
```

В примере выше мы не использовали атрибуты. Для атрибутов тегов XML имеются специальные классы `AttributesImpl` и `AttributesNSImpl`, методы которых описаны в табл. 9.101. Второй из этих классов является производным от первого и добавляет методы с буквой `Q` (от англ. *qualified* — уточненный). Объекты этих классов также понимают обычные для словарей методы и операции `copy()`, `get()`, `has_key()`, `items()`, `keys()`, `values()`.

Таблица 9.101. *Методы объектов классов* `AttributesImpl` *и* `AttributesNSImpl`

Метод	Описание
`getLength()`	Количество атрибутов
`getNames()`	Имена атрибутов

Таблица 9.101 *(окончание)*

Метод	Описание
`getType(name)`	Тип имени атрибута (обычно "`CDATA`")
`getValue(name)`	Значение атрибута с именем `name`
`getValueByQName(name)`	Значение уточненного имени `name`
`getNameByQName(name)`	Возвращает кортеж `(namespace, localname)`, т. е. пространство имен и локальное имя, для уточненного имени `name`
`getQNameByName(name)`	Возвращает уточненное имя по кортежу `(namespace, localname)`
`getQNames()`	Возвращает уточненные имена всех атрибутов

Пример в листинге 9.74 показывает использование пространств имен в SAX.

Листинг 9.74. Файл W3CExample.py. Пример с использованием пространств имен

```
import xml.sax, xml.sax.handler
from xml.sax.xmlreader import InputSource

class ContentHandler(xml.sax.handler.ContentHandler):
  def startElementNS(self, name, qname, attrs):
    print name, qname
    print attrs.items()
  def endElementNS(self, name, qname):
    print name, qname

input_source = InputSource()
input_source.setByteStream(open("W3CExample.xml", "r"))
xml_reader = xml.sax.make_parser()
xml_reader.setContentHandler(ContentHandler())
# включаем опцию namespaces
xml_reader.setFeature(xml.sax.handler.feature_namespaces, 1)
xml_reader.parse(input_source)
```

Пример, приведенный в листинге 9.75, взят из описания пространств имен в рекомендации консорциума W3C, доступной по адресу **href:http://www.w3.org/TR/REC-xml-names**.

Листинг 9.75. Файл W3CExample.xml. Пример из рекомендации W3C

```
<?xml version="1.0"?>
<!-- elements are in the HTML namespace, in this case by default -->
<html xmlns='http://www.w3.org/TR/REC-html40'>
  <head><title>Frobnostication</title></head>
  <body><p>Moved to
    <a href='http://frob.com'>here</a>.</p></body>
</html>
```

Результат выполнения:

```
('http://www.w3.org/TR/REC-html40', 'html') None
[]
('http://www.w3.org/TR/REC-html40', 'head') None
[]
('http://www.w3.org/TR/REC-html40', 'title') None
[]
('http://www.w3.org/TR/REC-html40', 'title') None
('http://www.w3.org/TR/REC-html40', 'head') None
('http://www.w3.org/TR/REC-html40', 'body') None
[]
('http://www.w3.org/TR/REC-html40', 'p') None
[]
('http://www.w3.org/TR/REC-html40', 'a') None
[((None, 'href'), 'http://frob.com')]
('http://www.w3.org/TR/REC-html40', 'a') None
('http://www.w3.org/TR/REC-html40', 'p') None
('http://www.w3.org/TR/REC-html40', 'body') None
('http://www.w3.org/TR/REC-html40', 'html') None
```

Как и положено по рекомендациям W3C, вложенные теги оказываются в заданном пространстве имен.

Замечание

На момент написания книги стандартный анализатор XML, использованный в Python, не выдавал `None` в качестве параметра `qname`. Возможно, это будет исправлено в будущих версиях.

Модуль *xml.parsers.expat*

Модуль `xml.parsers.expat` является интерфейсом к XML-анализатору Expat, включенному в стандартный набор модулей Python при посредстве модуля-расширения `pyexpat`. Expat является непроверяющим анализатором, т. е. он не тратит время на проверку соответствия документа своему DTD. Главное, что предоставляет модуль — функцию для порождения объекта типа `xmlparser`.

```
ParserCreate([enc[, ns_sep]]) -> xmlparser
```

где `enc` — кодировка документа. Поддерживаются кодировки: UTF-8, UTF-16, ISO-8859-1 (или latin-1) и ASCII; `ns_sep` — символ разделителя URI пространства имен и элемента, используемый при передаче имен элементов и атрибутов. Если задан, Expat будет принимать во внимание пространства имен.

В ходе работы XML-анализатор вызывает свои методы, которым перед началом работы необходимо присвоить функции с обработчиками соответствующих событий (табл. 9.103). Кроме того, `xmlparser`-объект имеет свои атрибуты, которые приведены в табл. 9.102.

Таблица 9.102. *Атрибуты `xmlparser`-объектов*

Атрибут	Описание
`Parse(data[, isfinal])`	Анализирует содержимое строки `data`, вызывая соответствующие обработчики. Истинный параметр `isfinal` указывает, что данный вызов анализатора — последний
`ParseFile(file)`	То же, но данные берутся из файлового объекта `file`
`SetBase(base)`	Устанавливает базовый URI для разрешения относительных URI в системных идентификаторах. Это значение передается приложению с параметром `base` функций `ExternalEntityRefHandler()`, `NotationDeclHandler()`, `UnparsedEntityDeclHandler()`
`GetBase()`	Возвращает строку с базовым URI (или `None`, если тот не был установлен)
`GetInputContext()`	Возвращает входные данные, которые вызвали обрабатываемое событие. Данные в кодировке той сущности, которая содержит текст
`ExternalEntityParserCreate(context[, enc])`	Создает дочерний анализатор для разбора внешней разобранной сущности
`ordered_attributes`	Флаг, указывающий, что атрибуты следует рассматривать не как словарь, а как список. В этом случае каждый атрибут будет представлен в виде двух последовательных элементов списка, а порядок следования атрибутов будет таким же, как XML-документе

Таблица 9.102 *(окончание)*

Атрибут	Описание
`returns_unicode`	Указывает на то, что обработчику будут передаваться Unicode-строки. Если флаг равен нулю, будут передаваться обычные строки в кодировке utf-8
`specified_attributes`	При истинном значении этого атрибута анализатор будет предлагать только те атрибуты, которые указаны в документе, а не те, которые указаны в описании атрибутов. По умолчанию атрибут содержит ложное значение

XML-анализатор при своей работе вызывает методы, описанные в табл. 9.102. Задать конкретные функции, которые будут выполнять методы, можно простым присваиваниям функций атрибутам.

Таблица 9.103. *Методы-обработчики XML*

Метод	Описание
`XmlDeclHandler(version, enc, standalone)`	Вызывается при анализе объявления XML (в самом начале XML-документа). Параметр `version` дает версию XML, `enc` — кодировку документа, `standalone` указывает на то, является ли документ одиночным (`1`), не является одиночным (`0`) или статус документа не был задан `-1`
`StartDoctypeDeclHandler (doctype, systemId, publicId, flag)`	Вызывается в начале обработки DTD документа. В качестве параметров передается тип документа, системный и общий идентификаторы, а `flag` указывает на наличие внутреннего блока определений
`EndDoctypeDeclHandler()`	Вызывается при завершении обработки DTD
`ElementDeclHandler(name, model)`	Вызывается для каждого объявления типа элемента. Указывается имя типа элемента и тип содержания элемента
`AttlistDeclHandler(elem, attr, type, default, req)`	Вызывается для каждого объявляемого в типе элемента атрибута. Передается имя элемента `elem`, имя атрибута `attr`, тип атрибута `type`, значение атрибута по умолчанию или `None` и флаг, указывающий на обязательность атрибута — `req`
`StartElementHandler (name, atts)`	Вызывается в начале каждого элемента. Обработчику передается имя элемента `name` и словарь с атрибутами `atts`
`EndElementHandler(name)`	Вызывается при закрытии элемента

Таблица 9.103 *(окончание)*

Метод	Описание
`ProcessingInstructionHandler(target, data)`	Вызывается для каждой инструкции обработки с указанием инструкции `data` для целевого приложения `target`
`CharacterDataHandler(data)`	Вызывается для символьных данных
`EntityDeclHandler(entity, pe, value, base, systemId, publicId, notation)`	Вызывается для всех объявлений сущностей. Аргумент `value` для сущностей-параметров и внутренних сущностей значением должен быть строкой с объявленным содержимым сущности. Для внешних сущностей `None`. Имя нотации задается параметром `notation` и будет `None` для обработанных сущностей. Параметр `pe` указывает, что сущность является параметром
`NotationDeclHandler(notation, base, systemId, publicId)`	Вызывается при встрече в XML-документе объявления нотации
`StartNamespaceDeclHandler(prefix, uri)`	Вызывается еще до `StartElementHandler` в случае, когда элемент содержит объявление пространства имен
`EndNamespaceDeclHandler(prefix)`	Вызывается при закрытии элемента, содержавшего объявление пространства имен
`CommentHandler(data)`	Вызывается для комментариев
`StartCdataSectionHandler()`	Вызывается в начале секции `CDATA`
`EndCdataSectionHandler()`	Вызывается в конце секции `CDATA`
`DefaultHandler(data)`	Вызывается для любых знаков, для которых обработчик не был задан
`DefaultHandlerExpand(data)`	То же, но перед его вызовом развертываются указатели внутренних сущностей
`NotStandaloneHandler()`	Вызывается для документа, который не был объявлен как одиночный. Если возвращает `0`, обработчик возбуждает исключение `XML_ERROR_NOT_STANDALONE`. Если этот обработчик не был задан, ошибки не происходит
`ExternalEntityRefHandler(context, base, systemId, publicId)`	Вызывается при встрече указателей внешних сущностей. Метод позволяет обрабатывать внешние сущности. Должен создать еще один обработчик с помощью `ExternalEntityParserCreate(context)`, инициализировав его соответствующими методами. Этот метод должен возвращать целое число. Если оно равно `0`, обработчик возбудит исключение `XML_ERROR_EXTERNAL_ENTITY_HANDLING`, в остальных случаях обработка XML-документа продолжится

Следующий пример показывает, как с помощью `xml.parsers.expat` извлечь из XML-документа все символы, находящиеся между тегами:

```
from xml.parsers.expat import ParserCreate
global cnt

cnt = 0
def myCDataHandler(data):
  global cnt
  cnt += 1
  print "%-3i: '%s'" % (cnt, data)

parser = ParserCreate()
parser.CharacterDataHandler = myCDataHandler
parser.Parse("""<?xml version="1.0" ?>
<tag>&quot;Example&quot;</tag>""")
```

Результат выполнения:

```
1  : '"'
2  : 'Example'
3  : '"'
```

Пакет *xml.dom*

В отличие от SAX, *DOM (Document Object Model, объектная модель документа)* предлагает для XML-документа конкретную объектную модель. Разница между SAX и DOM примерно такая же, как между катушечным магнитофоном и проигрывателем компакт-дисков: SAX ориентирован на последовательное чтение данных, тогда как DOM — на произвольный доступ к структуре документа.

Python версии 2.1 поддерживает DOM второго уровня (DOM level 2) согласно рекомендации консорциума W3C, см. **http://www.w3.org/TR/DOM-Level-2-Core/**.

DOM документ на языке XML представляется в виде дерева вложенных друг в друга объектов, в корне которого стоит объект, представляющий документ как целое, а ветвями являются все остальные объекты XML. То есть объекты находятся в узлах дерева и соединены дугами на основании отношения вложенности. По аналогии с деревом родства, узлы дерева находятся в "родственных" отношениях: *корневой узел (root node)* является *предком (ancestor)* всех остальных узлов, каждый из которых, соответственно, является его *потомком (descendant)*. Узел является *родителем (parent)* по отношению к своему непосредственному потомку, который в этом случае называется *ребен-*

ком (child) или *дочерним узлом (child node)*. Соответственно *дети (children)* одного родителя являются *братьями (siblings)* друг другу (на самом деле английское научное слово *sibling* означает брата или сестру). Кроме того, дети каждого родителя строго упорядочены от первого до последнего в порядке их следования в документе. Получается, что находясь в каком-либо узле дерева, с помощью методов объекта в этом узле мы можем двигаться в обоих направлениях по двум осям: ребенок—родитель и брат—брат. Более того, мы можем не только двигаться, но и произвольно изменять наше дерево, добавляя, перенося или удаляя узлы. Разумеется, ничто не мешает нам при работе с *DOM-деревом (DOM tree)* сохранять объекты в узлах под некоторыми именами с тем, чтобы позднее обратиться к конкретному узлу, ведь DOM-дерево предоставляет произвольный доступ к своим узлам.

Примечание

Оси ребенок—родитель брат—брат — не единственные оси, которые можно усмотреть в XML-документе. Например, возможна ось предыдущий—следующий для перехода от узла к узлу в порядке их следования в документе, что и происходит в одностороннем порядке при работе с SAX.

В качестве примера использования пакета `xml.dom` рассмотрим еще раз XML-документ в листинге 9.73 с точки зрения DOM.

Листинг 9.76. Файл DOM_view.py. Программа для вывода DOM-дерева

```
# импортируем модуль xml.dom.minidom
import xml.dom.minidom
# получаем корневой объект DOM-дерева
dom = xml.dom.minidom.parse("arith.xml")

# dom.normalize()

# все, что ниже, выводит DOM-дерево (используя отступы)
def output_tree(node, level=0):
  if node.nodeType == node.TEXT_NODE:
    print ". "*level, node.nodeType, `node.data`
  else:
    print ". "*level, node.nodeType, node.nodeName
  for child in node.childNodes:
    output_tree(child, level+1)

output_tree(dom)
```

Результат выполнения:

```
 9 #document
.  1 expression
. .  1 operation
. . .  3 u'\n'
. . .  3 u'  '
. . .  1 type
. . . .  3 u'+'
. . .  3 u'\n'
. . .  3 u'  '
. . .  1 operand
. . . .  3 u'2'
. . .  3 u'\n'
. . .  3 u'  '
. . .  1 operand
. . . .  3 u'\n'
. . . .  3 u'    '
. . . .  1 expression
. . . . .  1 operation
. . . . . .  3 u'\n'
. . . . . .  3 u'      '
. . . . . .  1 type
. . . . . . .  3 u'*'
. . . . . .  3 u'\n'
. . . . . .  3 u'      '
. . . . . .  1 operand
. . . . . . .  3 u'3'
. . . . . .  3 u'\n'
. . . . . .  3 u'      '
. . . . . .  1 operand
. . . . . . .  3 u'4'
. . . . . .  3 u'\n'
. . . . . .  3 u'    '
. . . .  3 u'\n'
. . . .  3 u'  '
. . .  3 u'\n'
```

С помощью рекурсивной функции `output_tree()` мы смогли обойти и вывести DOM-дерево с необходимыми отступами. Заметьте, что текстовые уз-

лы выдаются в виде Unicode-строк, причем, совсем необязательно, что соседние текстовые узлы сливаются в один. Раскомментируйте `dom.normalize()`, чтобы слить соседние текстовые узлы воедино. Для реализации примера из листинга 9.76 был использован `minidom` — достаточно "легкая" реализация DOM, входящая в стандартную поставку Python.

Модуль `xml.dom` содержит функции для регистрации реализации DOM, получения одной из доступных реализаций DOM, а также базовый класс `xml.dom.Node` и исключения, используемые при работе с DOM.

- `xml.dom.registerDOMImplementation(name, factory)`

  Функция регистрирует порождающую функцию `factory` под именем `name`, которая производит объекты, реализующие интерфейс `DOMImplementation` (см. ниже). В зависимости от реализации функция может возвращать один и тот же объект каждый раз.

- `xml.dom.getDOMImplementation([name[, features]])`

  Возвращает подходящую реализацию DOM. Например, для голой стандартной поставки Python это объект класса `xml.dom.minidom.DOMImplementation`. Можно указать требуемое имя реализации, например, `minidom`. Кроме того, можно указать список опций `features`, которые должна поддерживать реализация в виде последовательности кортежей `(опция, версия)`. Если аргументов не задано, может быть выдана любая из имеющихся реализаций DOM, но при этом будет учитываться значение переменной окружения `PYTHON_DOM`.

Как мы уже говорили выше, DOM-дерево состоит из узлов — объектов класса `Node`. В табл. 9.104 перечислены атрибуты—константы класса `Node` для обозначения типа узла, а в табл. 9.105 приведены методы класса `Node`.

Таблица 9.104. *Константы класса* `Node`

Атрибут	Значение	Описание
`ELEMENT_NODE`	1	Элемент
`ATTRIBUTE_NODE`	2	Узел с атрибутом
`TEXT_NODE`	3	Текстовый узел
`CDATA_SECTION_NODE`	4	Раздел `CDATA`
`ENTITY_REFERENCE_NODE`	5	Ссылка на сущность
`ENTITY_NODE`	6	Сущность
`PROCESSING_INSTRUCTION_NODE`	7	Инструкция по обработке
`COMMENT_NODE`	8	Комментарий
`DOCUMENT_NODE`	9	Документ

Таблица 9.104 *(окончание)*

Атрибут	Значение	Описание
DOCUMENT_TYPE_NODE	10	Тип документа
DOCUMENT_FRAGMENT_NODE	11	Фрагмент документа
NOTATION_NODE	12	Нотация

Таблица 9.105. *Методы и атрибуты объектов класса* `Node`

Метод или атрибут	Описание
nodeType	Целое число, обозначающее тип узла *(см. табл. 9.104)*
parentNode	Родительский узел текущего узла (`None` для корневого узла и для узлов класса `Attr`)
attributes	Объект класса `NamedNodeList` с атрибутами элемента (другие типы узлов содержат `None` в этом атрибуте)
previousSibling	Предыдущий дочерний узел того же родителя или `None`, если данный узел первый у своего родительского узла
nextSibling	Следующий дочерний узел или `None`
childNodes	Список дочерних узлов данного узла
firstChild	Первый дочерний узел данного узла или `None`
lastChild	Последний дочерний узел данного узла или `None`
localName	Строка с локальным именем (часть имени тега, стоящая после ":") или имя тега целиком
prefix	Часть имени тега перед ":", пустая строка или `None`
namespaceURI	Строка URI пространства имен, ассоциированного с именем элемента, или `None`
nodeName	Имя элемента. Имеет свой смысл для различных классов узлов, например, `tagName` для элементов и `name` для атрибутов. Может быть строкой или `None`, если узел не имеет такого свойства
nodeValue	Значение узла. Тоже имеет различный смысл в зависимости от класса узла. Например, это `data` текстового узла
hasAttributes()	Возвращает значение "истина", если узел имеет атрибуты
hasChildNodes()	Возвращает значение "истина", если узел имеет дочерние узлы
isSameNode(other)	Истина, если данный узел и узел `node` — один и тот же узел
appendChild(new)	Добавляет дочерний узел к концу дочерних узлов данного узла и возвращает `new`

Таблица 9.105 (окончание)

Метод или атрибут	Описание
`insertBefore (new, ref)`	Вставляет новый дочерний узел `new` перед существующим дочерним узлом `ref`, возвращая `new`. В случае, если `ref` не является дочерним узлом, возбуждается исключение `ValueError`
`removeChild(child)`	Удаляет дочерний узел `child` и возвращает его. Возбуждает `ValueError`, если `child` не принадлежит данному узлу
`replaceChild (new, old)`	Заменяет дочерний узел `old` на новый узел `new`. Узел `old` должен принадлежать данному узлу, иначе возбуждается `ValueError`
`normalize()`	Объединяет текстовые участки в единое целое, чтобы облегчить обработку XML-документа
`cloneNode (recursive)`	Клонирует данный узел. Указание `recursive` говорит о том, что и дочерние узлы необходимо клонировать

От базового класса `Node` происходят классы `ProcessingInstruction`, `Comment`, `Attr`, `Element`, `Document`, `DocumentType`, `DocumentFragment`, `CharacterData` (и его подклассы `Text`, `CDATASection`) (см. табл. 9.104). Атрибуты этих классов и будут разобраны ниже с той или иной степенью детализации.

Класс `Document` представляет XML-документ как единое целое и предоставляет несколько методов для создания узлов различных типов и поиска узлов по имени. Кроме того, методы этого класса имеют атрибут `documentElement` — корневой элемент данного документа. Методы объектов класса `Document` сведены в табл. 9.106. Заметьте, что они `create`-методы и не включают производимые ими объекты в DOM-дерево. Для этого применяются другие методы соответствующих `Node`-объектов.

Таблица 9.106. *Методы класса* `Document`

Метод	Описание
`createElement(name)`	Создает новый элемент с именем `name`
`createElementNS(namespaceURI, name)`	То же, но с пространством имен `namespaceURI`. Аргумент `name` может иметь префикс
`createTextNode(data)`	Создает текстовый узел, наполняя его данными `data`
`createComment(data)`	Создает узел с комментарием `data`
`createProcessingInstruction(target, data)`	Создает инструкцию `data` для целевого приложения `target`
`createAttribute(name)`	Создает атрибут с именем `name`

Таблица 9.106 *(окончание)*

Метод	Описание
`createAttributeNS (namespaceURI, qname)`	То же, но с использованием пространства имен `namespaceURI` и уточненного имени `qname`
`getElementsByTagName (name)`	Ищет по всему документу элементы с именем `name`, возвращая их список
`getElementsByTagNameNS (namespaceURI, localname)`	То же, но с учетом пространства имен. Здесь `localname` — часть имени элемента после префикса и `":"`

Объект класса `Document` имеет атрибут `doctype`, в котором содержится объект класса `DocumentType`. Он хранит атрибуты, относящиеся к определяемым в XML-документе нотациям и сущностям. В табл. 9.107 приведены атрибуты объектов этого класса.

Таблица 9.107. *Атрибуты объектов класса* `DocumentType`

Атрибут	Описание
`publicId`	Общий идентификатор
`systemId`	Системный идентификатор
`internalSubset`	Полное внутреннее подмножество документа (или `None`)
`name`	Название корневого элемента объявления типа документа
`entities`	`NamedNodeMap` внешних сущностей. Повторные объявления одной и той же сущности не учитываются. Может быть `None`, если XML-анализатор не дает необходимой информации или сущности не были определены
`notations`	То же для определений нотаций

Примечание

Модуль `xml.dom.minidom` из стандартной поставки Python не поддерживает разбор объявления типа документа, поэтому не все описанные в табл. 9.107 атрибуты будут доступны. Модулем также не поддерживаются интерфейсы: `CharacterData`, `CDATASection`, `Notation`, `Entity`, `EntityReference`, `{DocumentFragment}`. Они не так часто нужны при работе с XML.

Объекты класса `Comment` имеют атрибут `data`, в котором находится содержимое комментария. Объекты `ProcessingInstruction` помимо `data` содержат атрибут `target`, в котором указано целевое приложение для инструкции обработки.

Подробнее следует остановиться на классе Element, т. к. он вместе с Text обычно представлен в большинстве узлов DOM-дерева. В табл. 9.108 приведены методы объектов класса Element, дополнительные к методам объектов класса Node. Объекты класса Element также имеют атрибут tagName — строку с именем тега (при использовании пространств имен в tagName может присутствовать ":").

Таблица 9.108. *Методы объектов класса* Element

Метод	Описание
getElementsByTagName (tagName)	Ищет элементы с именем tagName, возвращая их список
getElementsByTagNameNS (tagName)	То же, но с учетом простанства имен
getAttribute(attrname)	Возвращает значение атрибута attrname в виде строки
getAttributeNS (namespaceURI, localname)	То же для заданного пространства имен
getAttributeNode (attrname)	Возвращает узел с атрибутом attrname (объект класса Attr)
getAttributeNodeNS (namespaceURI, localName)	То же для заданного пространства имен
removeAttribute (attrname)	Удаляет атрибут по имени attrname
removeAttributeNS (namespaceURI, localname)	То же, но с использованием пространства имен
removeAttributeNode (attrname)	То же, но возвращает Attr-объект, соответствующий удаленному атрибуту. Возбуждает исключение xml.dom.NotFoundErr, если атрибут отсутствует
setAttribute(attrname, value)	Устанавливает значение атрибута из строки
setAttributeNS(namespaceURI, qname, value)	То же для пространства имен namespaceURI и уточненного имени qname
setAttributeNode(new)	Добавляет атрибут к элементу, заменяя существующий в случае необходимости (старый атрибут возвращается методом). Если new уже используется, возбуждается исключение xml.dom.InuseAttributeErr
setAttributeNodeNS(new)	То же, но с учетом пространства имен

Наконец, узлы с текстом относятся к типу `Text` и имеют атрибут `data`. В отличие от элементов, они не могут иметь дочерних узлов. В принципе, классы `Text` и `CDATASection` отличаются только типом узла.

Пример в листинге 9.77 показывает построение DOM-дерева с нуля, только вызовами соответствующих методов.

Листинг 9.77. Построение DOM-дерева "вручную"

```
from xml.dom import minidom
dom = minidom.Document()
txt = dom.createElement("text")
dom.appendChild(txt)
comment = dom.createComment(" строим DOM ")
txt.appendChild(comment)
br = dom.createElement("br")
txt.appendChild(br)
textdata = dom.createTextNode(" текст ")
txt.appendChild(textdata)
print dom.toprettyxml()
```

Результат выполнения:

```
<?xml version="1.0" ?>
<text>
        <!-- строим DOM -->
        <br/>
         текст
</text>
```

Предупреждение

Русские буквы были использованы только для целей иллюстрации. Было бы более правильно назначить документу кодировку utf-8, а текст переводить из принятой кодировки в Unicode-строку. Правда, в этом случае вывести текст с помощью `toprettyxml()` будет невозможно, и нужно будет написать собственную функцию вывода.

9.15. Элементы мультимедиа

Конечно, *мультимедиа* (как многосредовое представление информации в человеко-машинных интерфейсах) — это слишком громко сказано. Набор модулей, которые оригинальная документация относит к этой категории,

занимается в основном звуковыми и графическими форматами. (Из всего набора модулей в данной книге разобраны только четыре: `wave`, `colorsys`, `imghdr` и `sndhdr`.) Табл. 9.109 дает представление о наборе модулей, которые документация по Python отнесла к этой категории.

Таблица 9.109. *Модули для работы с графикой и аудио*

Модуль	Описание
`aifc`	Чтение и запись аудиофайлов в форматах AIFF и AIFC
`audioop`	Обработка сырых аудиоданных
`chunk`	Чтение кусочных IFF-данных, использующееся для форматов AIFF, TIFF, RMFF
`colorsys`	Преобразование между цветовыми системами
`imageop`	Обработка сырых графических данных
`imghdr`	Определение типа графического файла
`rgbimg`	Чтение и запись аудиофайлов в формате SGI RGB
`sndhdr`	Определение типа аудиофайла
`sunau`	Чтение и запись аудиофайлов в формате AU
`wave`	Чтение и запись аудиофайлов в формате WAV

Тем, кому не хватает возможностей перечисленных модулей, можно порекомендовать пакет Python Imaging Library (PIL) *(см. разд. 12.1)*.

9.15.1. Модуль *wave*

Для работы с WAV-файлами стандартная библиотека Python предусматривает модуль `wave`. Как известно, типичный WAV-файл состоит из двух частей: заголовочной части с информацией о формате файла, количестве каналов, частоте и т. п. и собственно аудиоданных, которые идут сразу за заголовочной частью до конца файла.

Модуль `wave` предоставляет функцию.

```
open(f[, mode]) -> waveobj
```

```
openfp(f[, mode]) -> waveobj
```

Функция открывает файл `f` (строка с именем файла или файловый объект с возможностью `seek()`). Параметр `mode` указывает режим открытия: `r` и `rb` для чтения (в этом случае возвращается объект класса `wave.Wave_read`), `w` и `wb` для записи (`wave.Wave_write`).

В табл. 9.110 описываются методы объектов первого типа, а в табл. 9.111 — второго.

Таблица 9.110. *Методы для чтения WAV-файла*

Метод	Описание
`getnchannels()`	Возвращает число аудиоканалов (1 — моно, 2 — стерео)
`getsampwidth()`	Возвращает точность отсчета дискретизации в байтах
`getframerate()`	Возвращает частоту дискретизации в отсчетах в секунду
`getnframes()`	Возвращает общее число отсчетов в файле
`getcomptype()`	Возвращает тип использованного сжатия (пока поддерживается только `"NONE"`)
`getcompname()`	Возвращает описание использованного сжатия (пока поддерживается только `"not compressed"`)
`getparams() -> params`	Возвращает кортеж `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, названия соответствуют перечисленным выше `get`-методам
`readframes(n)`	Читает `n` отсчетов, не более, и возвращает строку байтов
`rewind()`	Возвращает указатель на начало аудиоданных
`setpos(pos)`	Устанавливает указатель на позицию `pos`
`tell() -> pos`	Возвращает текущую позицию указателя
`close()`	Закрывает файл

Таблица 9.111. *Методы для записи WAV-файла*

Метод	Описание
`setnchannels(n)`	Устанавливает число аудиоканалов
`setsampwidth(n)`	Устанавливает точность одного отсчета дискретизации
`setframerate(n)`	Устанавливает частоту отсчетов дискретизации
`setnframes(n)`	Устанавливает общее число отсчетов. Автоматически корректируется при закрытии файла
`setcomptype (type, name)`	Устанавливает тип и название используемого сжатия
`setparams (params)`	Устанавливает параметры из кортежа `params: (nchannels, sampwidth, framerate, nframes, comptype, compname)`
`tell() -> pos`	Возвращает текущую позицию указателя
`writeframesraw (data)`	Записывает аудиоданные в файл без корректировки `nframes`. После применения этого метода использование `set`-методов возбуждает исключение

Таблица 9.111 *(окончание)*

Метод	Описание
`writeframes (data)`	То же, но обеспечивает правильность значения `nframes`
`close()`	Корректирует `nframes` должным образом и закрывает файл

Следующий пример показывает как создать WAV-файл с синусоидальным сигналом частотой 1 кГц и продолжительностью 1 с.

Листинг 9.78. Файл sine_wave.py. Синусоидальный сигнал

```
import wave, math, struct
duration = 1.0           # продолжительность
frequency = 1000.0       # частота сигнала
A = 10000.0              # амплитуда
framerate = 22050        # частота дискретизации, Гц

wf = wave.open("sine.wav", "wb")
wf.setnchannels(1)       # моно
wf.setsampwidth(2)       # 2 байта (16-разрядный звук)
wf.setframerate(framerate)

omega = 2*math.pi*frequency
for i in xrange(duration*framerate):
  t = float(i)/framerate     # точка очередного отсчета
  y = A * math.sin(omega*t)  # амплитуда сигнала в этом отсчете
  iy = int(y)
  wf.writeframesraw(chr(iy&0xFF) + chr((iy>>8)&0xFF))
wf.close()
```

9.15.2. Модуль *colorsys*

Модуль `colorsys` содержит функции для преобразования между различными цветовыми системами координат. В табл. 9.112 представлены поддерживаемые модулем цветовые модели.

Таблица 9.112. *Цветовые модели*

Сокращение	Расшифровка
RGB	Red, Green, Blue (красный, зеленый, синий)

Таблица 9.112 *(окончание)*

Сокращение	Расшифровка
YIQ	Система, используемая в композитных видеосигналах (Y несет компоненту яркости, а I и Q — задают цветность)
HLS	Hue, Luminance, Saturation (оттенок, яркость, насыщенность)
HSV	Hue, Saturation, Value (оттенок, насыщенность, величина)

Функции преобразования следующие (все входные и выходные значения лежат между 0.0 и 1.0).

- `hls_to_rgb(h, l, s) -> r, g, b`
- `hsv_to_rgb(h, s, v) -> r, g, b`
- `rgb_to_hls(r, g, b) -> h, l, s`
- `rgb_to_hsv(r, g, b) -> h, s, v`
- `rgb_to_yiq(r, g, b) -> y, i, q`
- `yiq_to_rgb(y, i, q) -> r, g, b`

В модуле также определены константы: `ONE_SIXTH`, `ONE_THIRD` и `TWO_THIRD`.

9.15.3. Модули *sndhdr* и *imghdr*

Иногда бывает очень важно определить, что находится в том или ином файле. Для распознавания формата аудио- и (растровых) графических файлов можно применить модули `sndhdr` и `imghdr` соответственно. Работать с этими модулями достаточно легко: нужно просто указать имя файла определенной в модуле функции `what`.

Для графических файлов функция `imghdr.what()` возвращает название формата. В Python 2.1 эта функция распознает следующие форматы: `'rgb'` (SGI ImgLib файл), `'gif'` (GIF 87a и 89a), `'pbm'`, `'pgm'`, `'ppm'`, `'tiff'`, `'rast'`, `'xbm'`, `'jpeg'` (JPEG данные в формате JFIF), `'bmp'`, `'png'`. Функция заглядывает внутрь файла, поэтому для правильноой работы расширение файла ей не нужно:

```
>>> import imghdr
>>> print imghdr.what("clover.jpg")  # JPEG-файл
jpeg
>>> print imghdr.what("screen.gif")  # GIF-файл
gif
>>> print imghdr.what("clover.unk")  # файл с непонятным расширением
jpeg
```

Аналогично работает функция sndhdr.what для распознавания формата звукового файла. Она возвращает кортеж из пяти элементов: (тип, отсчетов_в_секунду, число_каналов, отсчетов, битов_на_отсчет). Тип может быть одним из: 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', 'ul'. Число отсчетов_в_секунду может быть нулем, если соответствующую информацию трудно получить из файла. Аналогично число_каналов. Количество отсчетов дискретизации либо отражает реальное значение, либо равно -1, если соответствующее значение было трудно определить. И, наконец, число битов_на_отсчет может либо содержать реальную точность отсчетов дискретизации, либо равняться A или U для обозначения нелинейной A- и μ-дискретизации.

Следующий пример иллюстрирует работу функции:

```
>>> import sndhdr
>>> print sndhdr.what("activate.wav")
('wav', 44100, 1, -1, 16)
>>> print sndhdr.what("audiotest.au")
('au', 8000, 1, 23461, 'U')
```

В первом случае мы имеем монофонический WAV-файл с частотой дискретизации 44,1 кГц, точностью дискретизации 16 битов на отсчет, общее число отсчетов неизвестно. Во втором — файл в формате AU с частотой 8 кГц и 23 461 отсчетов, закодированных с применением μ-закона.

9.16. Элементы криптографии

В этом разделе собраны функции, которые относятся к процедурам поддержки аутентификации по паролю, проверке целостности данных и, собственно, шифрованию и дешифрованию данных. Кроме описываемых в этом разделе модулей md5, rotor и crypt, стандартная библиотека Python содержит модуль sha, предоставляющей еще один способ для дайджеста данных, помимо md5-сумм.

9.16.1. Модуль *md5*

Модуль md5 является интерфейсом к алгоритму свертки (дайджеста) сообщений, описанному в RFC-1321. С помощью md5 можно считать контрольные суммы некоторых данных (например, на FTP-серверах часто вместе с файлами лежит файл с md5-суммами). При изменении в данных хотя бы на один бит их md5-сумма неузнаваемым образом меняется. Еще одно применение md5 связано с (односторонним) шифрованием паролей при авторизации пользователя. По паролю, введенному пользователем, вычисляется его md5-сумма и сравнивается с хранимой в системе настоящей суммой. В слу-

чае их совпадения пароль считается введенным правильно. Еще одно применение дайджестов — индексирование наборов данных. Например, мы можем с большой точностью (и очень малой вероятностью ошибки) найти на жестком диске два одинаковых файла, если найдем два файла с одинаковыми md5-суммами.

Модуль md5 позволяет порождать md5-объект с помощью функции md5.new. Необязательным аргументом этой функции может быть строка. В дальнейшем свертываемое сообщение можно дополнять с помощью метода update. Методы md5-объекта перечислены в табл. 9.113.

Таблица 9.113. *Методы* `md5`*-объекта*

Метод	Описание
`copy()`	Создает копию `md5`-объекта
`digest()`	Возвращает строку с дайджестом. Строка из шестнадцати байтов, которая может содержать любые символы (в том числе и нулевые байты)
`hexdigest()`	Возвращает строку с шестнадцатеричным представлением дайджеста
`update(s)`	Добавляет строку `s` к накопленному сообщению

Модуль md5 также содержит объект типа md5.MD5Type для проверки принадлежности объектов к md5-объектам.

```
>>> import md5
>>> print md5.new("abcd").hexdigest()
e2fc714c4727ee9395f324cd2e7f331f
>>> m = md5.new("a")
>>> m.update("b")
>>> m.update("cd")
>>> print m.hexdigest()
e2fc714c4727ee9395f324cd2e7f331f
```

В этом примере мы получаем шестнадцатеричное представление md5-суммы для строки "abcd". Первый раз мы вводим строку целиком, а затем ее же по частям.

Стандартная библиотека модулей имеет также модуль sha для другого вида дайджеста — SHA (Secure Hash Standard, стандарт безопасного хэша). Он имеет тот же самый интерфейс, что и рассмотренный здесь модуль md5.

Еще один распространенный метод для проверки целостности данных называется *CRC (Cyclic Redundancy Check, циклический избыточный контроль)* и

употребляется в модулях zipfile, zlib, binascii. В отличии от md5 или sha, этот вид дайджеста не является криптографически стойким.

9.16.2. Модуль *rotor*

Модуль rotor реализует алгоритм шифрования, сходный с тем, что использовался в шифровальной машине "Enigma" в годы второй мировой войны. Не вникая в детали метода, следует сказать, что алгоритм, примененный в данном модуле, не является достаточно сильным криптографическим методом для хранения важных тайн.

```
rotor.newrotor(key[, numrotors]) -> rotor_obj
```

где key — шифровальный ключ; numrotors — количество перестановок роторов, по умолчанию 6; rotor_obj — rotor-объект.

В табл. 9.114 перечислены методы rotor-объекта. Далее под plaintext понимается шифруемая строка, а под ciphertext — строка — результат шифрования.

Таблица 9.114. *Методы rotor-объекта*

Метод	Описание
setkey(key)	Устанавливает ключ key в качестве ключа для шифрования
encrypt(plaintext) -> ciphertext	Помещает rotor-объект в исходное состояние и шифрует заданную строку
encryptmore(plaintext) -> ciphertext	Продолжает шифрование строк
decrypt(ciphertext) -> plaintext	Помещает rotor-объект в исходное состояние и дешифрует заданную строку
decryptmore(ciphertext) -> plaintext	Продолжает дешифрацию данных

В следующем примере (листинг 9.79) мы шифруем текст в два приема, а дешифруем за один раз.

Листинг 9.79. Файл cipher.py. Шифрование текста

```
from rotor import newrotor
rot = newrotor('Python в подлиннике', 20)
text = """Модуль rotor не является точной алгоритмической моделью
машины Enigma, т. к. имеет произвольное число роторов и позволяет
шифровать произвольные 8-разрядные данные, а не только латинские буквы.
```

```
Кроме того, при инициализации роторов используется датчик
псевдослучайных чисел."""
# шифруем:
cif = rot.encrypt("Дополнение. ")
cif = cif + rot.encryptmore(text)
# дешифруем:
print newrotor('Python в подлиннике', 20).decrypt(cif)
```

9.16.3. Модуль *crypt*

Модуль `crypt` (доступен в UNIX) предоставляет возможность для шифрования пароля, методом, основанным на DES (Data Encryption Standard, (американский) стандарт шифрования данных).

Замечание

Не сто́ит применять DES для чего-либо помимо аутентификации по паролю.

```
crypt.crypt(key, salt) -> coded
```

где `key` — пароль, введенный пользователем; `salt` — "соль", двухсимвольная строка для указания одной из 4096 комбинаций алгоритма; `coded` — закодированный пароль, состоящий из тех же символов, что и `salt`.

Алфавит, используемый для `salt`, состоит из 64 символов: строчные и прописные буквы, цифры, символы "." и "/".

Пример в листинге 9.80 показывает применение `crypt` для аутентификации пользователя.

Листинг 9.80. Проверка пароля

```
from crypt import crypt
from getpass import getpass
passwords = {  # словарь с (зашифрованными) паролями пользователей
  'user': '12J6dpXX4Pqxw'
}

login = raw_input("Login: ")
password = getpass("Password: ")

if passwords.has_key(login):  # если пользователь существует
  pwd = passwords[login]
  salt = pwd[:2]
```

```
  if crypt(password, salt) == pwd:  # проверяем пароль
    print "Welcome!"
  else:
    print "Login incorrect."
else:
  print "Login incorrect."
```

9.17. Защищенная среда исполнения: модуль *rexec*

В некоторых случаях необходимо запускать программы, полученные из ненадежных источников в безопасном для запускающего их пользователя режиме. Модуль `rexec` позволяет создавать и конфигурировать контролируемую среду исполнения программ. Исполняемые при помощи `rexec` программы ограничиваются в наборе загружаемых модулей и вызываемых функций и возможностях ввода/вывода, но не ограничиваются по объему занимаемой памяти и процессорного времени. Последние ограничения нужно задавать уже средствами операционной системы.

Для обеспечения защищенной среды исполнения используется класс `RExec`, конструктор которого описан ниже.

```
RExec ([hooks[, verbose]]) -> rexec_obj
```

где `hooks` — экземпляр класса `RHooks` (или производного от него класса). По умолчанию (если `hooks` не указан или `None`) используется экземпляр стандартного `RHook` класса; `verbose` — если "истина", на стандартный вывод будет печататься дополнительная отладочная информация; `rexec_obj` — экземпляр класса `RExec`.

В табл. 9.115 приведены и объяснены атрибуты класса `RExec`, а в табл. 9.116 — его методы.

Таблица 9.115. *Атрибуты класса* `RExec`

Атрибут	Описание	Значение по умолчанию
`nok_builtin_names`	Список запрещенных встроенных имен	`('open', 'reload', '__import__')`
`ok_builtin_modules`	Список безопасных стандартных модулей	`('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'pcre', 'rotor', 'select', 'strop', 'struct', 'time')`

Таблица 9.115 *(окончание)*

Атрибут	Описание	Значение по умолчанию
`ok_path`	Список каталогов, в которых производится поиск модулей при импорте в условиях ограниченной среды исполнения	Что и `sys.path`
`ok_posix_names`	Список доступных функций модуля `os`	`('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'getuid', 'getgid', 'geteuid', 'getegid')`
`ok_sys_names`	Список доступных функций модуля `sys`	`('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint')`

Таблица 9.116. *Методы объектов класса* `RExec`

Метод	Описание
`r_eval(code)`, `s_eval(code)`	Вычисляет значение выражения `code`, заданного кодовым объектом или строкой
`r_exec(code)`, `s_exec(code)`	Исполняет код `code`, заданный кодовым объектом или строкой
`r_execfile(filename)`, `s_execfile(filename)`	Исполняет программу в файле `filename`
`r_import(modulename[, globals [, locals[, fromlist]]])`, `s_import(modulename[, globals [, locals[, fromlist]]])`	Импортирует модуль `modulename`
`r_open(filename[, mode [, bufsize]])`, `s_open(filename [, mode[, bufsize]])`	Открывает файл, возвращая файловый объект. По умолчанию любой файл можно прочитать, но запись в файл запрещена
`r_reload(module)`, `s_reload(module)`	Перезагружает указанный модуль `module`
`r_unload(module)`, `s_unload(module)`	Выгружает модуль, снимая его из словаря `sys.modules` внутри контролируемой среды исполнения

Разница между `r_`-методами и `s_`-методами в том, что последние предоставляют доступ к стандартным потокам ввода/вывода. При конфигурировании

собственной среды исполнения все эти методы можно переопределять, но только путем наследования от класса `Rexec`. Оригинальные встроенные функции, описанные в *статье "Функции компиляции и исполнения" разд. 8.6.1*, заменяются методами, приведеными в табл. 9.116.

В листинге 9.81 дописывает восклицательный знак ко всем файлам в каталоге `sandbox`. Если мы хотим быть уверены в том, что она не повредит никакие другие файлы, мы можем запускать ее в контролируемой среде исполнения, пример которой приведен в листинге 9.82. Для испытания программы необходимо создать каталог `sandbox` и поместить туда один или несколько файлов, после чего запустить `safe_env.py`. С помощью стандартного класса `RExec` запустить программу `no_trust.py` не удается, т. к. стандартный `RExec` не дает открывать файлы для изменения. Зато созданный на основе `RExec` класс `MyRExec` позволяет обращаться к файлам, находящимся в каталоге, путь к которому начинается с `sandbox`.

Листинг 9.81. Файл no_trust.py. "Неблагонадежная" программа

```
"""Внимание! Программа дописывает восклицательный знак в каждый файл
в каталоге sandbox"""

import glob
i = 0
for file_name in glob.glob("sandbox/*"):
  try:
    f = open(file_name, "a")
    f.write("!")
    f.close()
    i = i + 1
  except:
    pass
print i, "file(s) changed"
```

Листинг 9.82. Файл safe_env.py. Среда для безопасного исполнения

```
import rexec, os, string
class MyRExec(rexec.RExec):
  """Среда контролируемого исполнения. По сравнению со стандартным
  RExec: изменен метод r_open. Пример основан на документации по
  Python.
  """
```

```
  def r_open(self, file, mode='r', buf=-1):
    if mode in ('r', 'rb'):
      pass
    elif mode in ('w', 'wb', 'a', 'ab'):
      if string.find(file, 'sandbox' + os.sep) != 0:
        raise IOError, "can't write outside sandbox"
      elif string.find(file, os.pardir) >= 0:
        raise IOError, "parent dir access forbidden"
    else:
      raise IOError, "Illegal open() mode"
    return open(file, mode, buf)

rexec.RExec().r_execfile("no_trust.py")
MyRExec().r_execfile("no_trust.py")
```

Результат выполнения:

```
0 file(s) changed
1 file(s) changed
```

9.18. Python о себе

Python является языком программирования, хорошо осведомленным о своем внутреннем устройстве. Описанные в этом разделе модули — лишь часть стандартных модулей, служащих этой цели. Так как наша книга не ставит целью столь глубоко изучать внутренние механизмы Python (хотя они стоят того!), интересующиеся могут почитать оригинальную документацию. Табл. 9.117 позволит вам сориентироваться в модулях, относящихся к лексическому и синтаксическому анализатору, дизассемблеру и другим интересным утилитам.

Таблица 9.117. *Модули для работы над самим языком Python*

Модуль	Описание
`dis`	Дизассемблер для байт-кода *(см. разд. 9.18.3)*
`keyword`	Содержит список зарезервированных слов языка Python *(см. разд. 9.18.1)*
`parser`	Обеспечивает интерфейс к деревьям синтаксического разбора (AST)
`py_compile`	Компилирует файл с исходным кодом в файл с байт-кодом *(см.. разд. 9.18.2)*

Таблица 9.117 *(окончание)*

Модуль	Описание
`compileall`	То же для всех файлов в каталогах `sys.path` *(см. разд. 9.18.2)*
`pyclbr`	Поддерживает извлечение информации, необходимой для инструментов инспекции классов и функций в исходном коде (такие инструменты по-английски называются *class browser*)
`symbol`	Содержит константы, представляющие внутренние узлы деревьев разбора
`tabnanny`	Утилита для обнаружения проблем с отступами, возникающими при смешении табуляций и пробелов
`tokenize`	Лексический сканер исходного Python-кода
`token`	Содержит константы, представляющие терминальные узлы деревьев разбора

Кстати, файл с грамматикой Python называется `Grammar` и находится в архиве с исходным кодом в каталоге `Grammar`.

Чтобы удовлетворить любопытство, следующий пример приводит дерево грамматического разбора для выражения `2+2`:

```
>>> import parser, pprint
>>> tree = parser.expr("2+2")
>>> pprint.pprint(tree.totuple())
(258,
 (312,
  (291,
   (292,
    (293,
     (294,
      (296,
       (297,
        (298,
         (299,
          (300,
           (301, (302, (303, (304, (2, '2')))))),
           (14, '+'),
           (301, (302, (303, (304, (2, '2'))))))))))))))),
 (4, ''),
 (0, ''))
```

9.18.1. Модуль *keyword*

В модуле `keyword` хранится информация о зарезервированных словах. Эти слова не могут употребляться в качестве идентификаторов в программах на Python.

```
iskeyword(s) -> 0, 1
```

Возвращает значение "истина", если в строке `s` указано ключевое слово и "ложь" в противном случае.

Также в модуле есть список ключевых слов `keyword.kwlist` и словарь `keyword.kwdict`, содержащий ключевые слова в качестве ключей.

9.18.2. Модули *py_compile* и *compileall*

Модуль `py_compile` содержит единственную функцию `py_compile.compile()` для компиляции исходного кода на языке Python в байт-код Python. О механизме автоматической компиляции модулей *(см. статью "Оператор import" в разд. 8.5.6)*.

```
py_compile.compile(file[, cfile[, dfile]])
```

где `file` — имя файла с исходным кодом; `cfile` — имя файла с байт-кодом. По умолчанию: `file + "c"` или `file + "o"` (если интерпретатор запущен в режиме оптимизации, т. е. с ключами `-O` или `-OO`); `dfile` — имя файла для использования в сообщениях об ошибках. По умолчанию используется `file`.

Данная функция обычно применяется для компилирования модулей, как и функции из модуля `compileall`, описанные ниже.

Для компилирования всех модулей в каталоге применяется следующая функция.

```
compileall.compile_dir(dir[, maxlevels[, ddir[, force]]])
```

где `dir` — каталог с модулями; `maxlevels` — глубина погружения в подкаталоги (по умолчанию `10`); `ddir` — указывает базовый путь, который будет использоваться в сообщениях об ошибках (по умолчанию — `dir`); `force` — флаг, при истинном значении которого модули перекомпилируются даже если (судя по времени создания файлов) они актуальны.

А следующая ниже функция позволяет перекомпилировать модули, находящиеся в каталогах, прописанных в списке `sys.path`.

```
compileall.compile_path([skip_curdir[, maxlevels[, force]]])
```

где `skip_curdir` — не компилировать модули в текущем каталоге (по умолчанию `1`); `maxlevels` — глубина погружения в подкаталоги (по умолчанию `0`); `force` — флаг принудительной перекомпиляции (по умолчанию тоже `0`).

Следующая команда часто применяется после внесения изменений в модули в текущем каталоге. В примере ниже команда подается пользователем root UNIX-системы:

```
# python -c 'import compileall; compileall.compile_dir(".")'
Listing . ...
Compiling ./02-001.py ...
Compiling ./02-002.py ...
Compiling ./02-003.py ...
...
# python -O -c 'import compileall; compileall.compile_dir(".")'
...
```

9.18.3. Модуль *dis*

Как уже было сказано ранее, перед исполнением программы Python компилирует ее в промежуточный код, называемый *байт-кодом (byte-code)*. Модуль dis позволяет получить расшифровку байт-кода в виде мнемонической нотации, напоминающей ассемблер. То есть модуль dis предоставляет функции *дизассемблера*. Следующий пример проясняет ситуацию:

```
import dis
code = """a+b"""
dis.disassemble(compile(code, '', 'eval' ))
```

Результат выполнения:

```
          0 SET_LINENO               0
          3 LOAD_NAME                0 (a)
          6 LOAD_NAME                1 (b)
          9 BINARY_ADD
         10 RETURN_VALUE
```

В этом примере участвует функция dis.disassemble(). А в следующем примере показано применение функции dis.distb() для дизассемблирования трассировочного объекта (по умолчанию берется объект, оставшийся от последнего исключения):

```
>>> 0/0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> import dis
>>> dis.distb()
          0 SET_LINENO               0
```

```
          3 SET_LINENO               1
          6 LOAD_CONST               0 (0)
          9 LOAD_CONST               0 (0)
-->       12 BINARY_DIVIDE
          13 PRINT_EXPR
          14 LOAD_CONST              1 (None)
          17 RETURN_VALUE
```

Из столь подробного вывода становится абсолютно ясно, какая инструкция возбудила исключение.

9.19. Графический интерфейс. Python и Tk

Для построения графического интерфейса в программах на языке Python могут применяться различные библиотеки. В табл. 9.118 приведены основные из них.

Таблица 9.118. *Библиотеки компонентов графического интерфейса (разного уровня)*

Название	Где можно найти	Характеристика	Платформы
Tkinter	Стандартный модуль	Многоплатформенный, довольно мощный, хорошее управление расположением компонентов; нет стандартного drag-n-drop, выглядит "неродным"	MUW
wxPython	**http://www.wxpython.org/**	Выглядит родным, стремится к многоплатформенности, активно совершенствуется, поддержка GL; пока есть не для всех платформ	UW
WPY	**ftp://ftp.python.org/pub/python/wpy/**	Подобие MFC; использует win32s (под Windows) и Tk (под UNIX)	UW
OpenGL	**http://starship.python.net/~da/PyOpenGL**	Поддержка OpenGL	UW
MacOS Tool Box	**ftp://ftp.python.org/pub/python/mac**	Входит в MacPython; только для Macintosh	M
PyGTK	**http://www.daa.com.au/~james/pygtk/**	Поддерживает только GTK+ и Gnome, богатый набор компонентов; родной только для GTK	U

Таблица 9.118 *(окончание)*

Название	Где можно найти	Характеристика	Платформы
PyQT/PyKDE	**http://www.thekompany.com/projects/pykde/?dhtml_ok=0**	Достаточно добротный набор для тех, кто использует Qt (под UNIX или Windows) и KDE	UW
Pythonwin	**http://www.python.org/windows/pythonwin/**	Построен вокруг MFC, поставляется вместе с оболочкой в пакете win32all; только для Windows	W
pyFLTK	**http://www.netpedia.net/hosting/fltk/**	Аналог Xforms, поддержка OpenGL, пока не закончен	UW
FXPy	**http://fxpy.sourceforge.net/**	Привязки к GUI-библиотеке FOX	UW
AWT, JFC, Swing	**Вместе с Jython**	Для Jython доступны средства, которые использует Java	Java

На самом деле, привязок к различным GUI-библиотекам несколько больше, интересующиеся могут посмотреть **http://starbase.neosoft.com/~claird/comp.lang.python/python_GUI.html** или с помощью поискового сервера найти для интересующей их GUI-библиотеки привязки к Python.

GUI-библиотеки могут быть многоуровневыми. Например, WxPython на платформе Linux базируется на многоплатформенной GUI-библиотеке wxWindows, которая, в свою очередь, базируется на GTK+ или на Motif, а те, в свою очередь, используют для вывода X Window. Кстати, для Motif в Python имеются свои привязки.

Этот раздел написан на основе строк документации внутри модуля `Tkinter` и с использованием отличного руководства "An Introduction to Tkinter" ("Введение в Tkinter"), написанного Фредриком Лундом, см. по адресу **http://www.pythonware.com/library/tkinter/introduction/**.

9.19.1. Классы модуля *Tkinter*

В набор стандартных модулей, поставляемых вместе с Python, входит модуль `Tkinter`. Этот модуль является оберткой для Tcl/Tk. (Кстати, язык *Tcl* и графический инструментарий к нему, Tk, также входят в поставку Python). `Tkinter` предоставляет визуальные компоненты и среду для исполнения событийно-управляемых программ с графическим пользовательским интерфейсом. Программы, построенные на Tk, являются многоплатформенными. Они одинаково выглядят на платформах X Window/UNIX, Windows и Macintosh.

В табл. 9.119 перечислены классы модуля `Tkinter`, а атрибуты и работу с наиболее типичными из них мы рассмотрим ниже. В `Tkinter` классы визуальных компонентов *(виджетов)* не являются наследниками друг от друга, как это делается в некоторых других инструментальных библиотеках, однако пользователь, разумеется, волен наследовать свои классы от предоставляемых `Tkinter`.

Таблица 9.119. *Классы модуля* `Tkinter`

Объект	Назначение
`Button`	Кнопка. Простая кнопка для выполнения команды и других действий
`Canvas`	Рисунок. Может использоваться для вывода графических примитивов, например, для построения графиков
`Checkbutton`	Флажок. Кнопка, имеющая два состояния, при нажатии изменяет состояние на противоположное
`Entry`	Поле ввода текста
`Frame`	Рамка. Содержит в себе другие визуальные компоненты
`Label`	Этикетка. Показывает некоторый текст или графическое изображение
`Listbox`	Список. Показывает список, из которого пользователь может выделить один или несколько элементов
`Menu`	Меню. Служит для организации всплывающих (popup) и ниспадающих (pulldown) меню
`Menubutton`	Кнопка-меню. Используется для организации ниспадающих меню
`Message`	Сообщение. Аналогично `Label`, но позволяет заворачивать длинные строки и легко меняет свой размер
`Radiobutton`	Переключатель. Представляет одно из альтернативных значений некоторой переменной. Обычно действует в группе. При нажатии на одну из кнопок, все остальные кнопки автоматически "отскакивают", совсем как кнопки выбора волны на радиоприемнике
`Scale`	Шкала. Позволяет задать числовое значение путем перемещения движка
`Scrollbar`	Полоса прокрутки. Может использоваться вместе с некоторыми другими компонентами для их прокрутки
`Text`	Форматированный текст. Позволяет показывать, редактировать и форматировать текст с использованием различных стилей, а также внедрять в текст рисунки и окна
`Toplevel`	Окно верхнего уровня. Показывается как отдельное окно. Содержит другие визуальные компоненты

На рис. 9.3 приведен пример окна, где представлены некоторые из перечисленных выше виджетов.

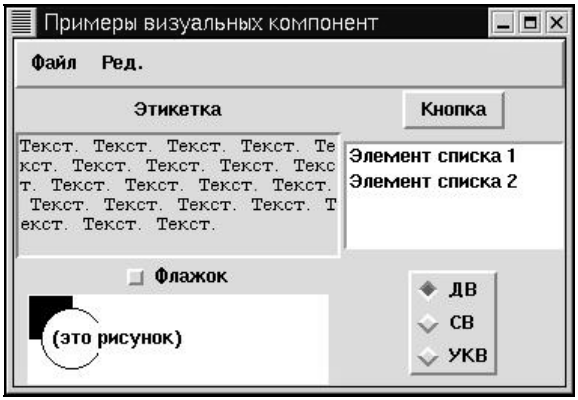


Рис. 9.3. `Tkinter` предоставляет целый ряд виджетов

К собственным методам виджетов `Tkinter` примешивает некоторые общие для всех или части виджетов методы, делегируемые специальным классам:

- класс `Misc` содержит более 150 методов, обеспечивающих различные сервисы для всех виджетов и главного окна. Этот класс — основной источник методов, имеющихся у виджетов;
- класс `Wm` примешивается к виджетам, которые могут действовать как окна, обеспечивая их методами для общения с оконным менеджером;
- класс `Grid` содержит методы *менеджера расположения (geometry manager)* типа "сетка". Организует виджеты в двумерную сетку (подобно тому, как задается таблица в языке HTML);
- класс `Pack` содержит методы менеджера расположения "упаковщик". При этом типе расположения виджеты укладываются в рамку родительского виджета с заданных сторон;
- класс `Place` содержит методы для координатного размещения виджетов. С помощью менеджера этого типа можно задавать абсолютные или относительные (относительно другого виджета) координаты и размеры виджетов.

9.19.2. Виджеты в общем

Прежде чем рассматривать конкретные виджеты, будет полезно узнать общие методы, свойственные всем виджетам.

Далее мы рассмотрим некоторые методы наиболее типичных классов виджетов. Так как каждый виджет имеет более 150 методов, то описать их все в

рамках данной книги не представляется возможным. Заинтересованные читатели могут обратиться к оригинальной документации по модулю `Tkinter` или языку Tcl/Tk или изучить виджеты, пользуясь диалоговым режимом, как это описано в *разд. 7.6.*

Создание виджетов

Виджеты любого класса *создаются* вызовом конструктора этого класса по следующей схеме.

```
Widget([master[, option=value, ...]]) -> widget
```

где `Widget` — название класса виджета; `master` — родительский виджет, к которому данный виджет будет относиться; `option` — конфигурационная опция (их может быть несколько); `value` — значение конфигурационной опции; `widget` — объект-виджет.

Базовые методы виджетов

Виджеты имеют большое количество методов. Чтобы их рассмотреть, мы разобьем их на несколько групп в соответствии с характером применения. Сразу же необходимо заметить, что часть методов, формально принадлежащих виджетам, относится скорее к самому интерфейсу Tk. Это нужно для того, чтобы при обработке событий (а обработчики событий обычно получают виджет, с которым событие связано) можно было влиять на глобальную среду существования виджетов, т. е. на сам `Tkinter`.

Замечание

Так как мы будем говорить о методах объектов, местоимение "сам" ("себя") будет употребляться по отношению к объекту, чей метод вызывается. Под "данным виджетом" будет подразумевается сам виджет, если другого виджета не задано в параметрах метода.

Управление событиями. Как уже говорилось, графический интерфейс пользователя требует от программы иметь событийную ориентацию. Ниже мы рассмотрим методы для управления *циклом событий* (табл. 9.120), *потоками событий* (табл. 9.121) и *ожиданием событий* (табл. 9.122).

Таблица 9.120. *Методы управления циклом событий*

Метод	Описание
`mainloop()`	Вхождение в главный цикл обработки событий. Может вызываться и из обработчиков событий, т. е. быть вложенным
`quit()`	Выход из главного цикла обработки событий

Таблица 9.120 *(окончание)*

Метод	Описание
`update()`	Обрабатывает все стоящие в очереди события, вызывая необходимые обработчики, завершает все задерживаемые перерисовки, вызывает все очередные задачи, выполняющиеся во время простоя (idle tasks). Внимание! Этот метод следует использовать с осторожностью, его нельзя вызывать из обработчиков событий и потому нельзя помещать в функции, которые могут вызывать обработчики событий
`update_idletasks()`	Вызывает выполнение всех задач времени простоя (idle tasks) без обработки других событий. Этот метод может использоваться для перерисовки окон после изменения содержимого

Отдельный виджет и его дочерние виджеты могут перехватить поток событий.

Таблица 9.121. *Методы управления потоком событий*

Метод	Описание
`grab_current()`	Возвращает виджет, на который направлен поток событий в этом приложении (или `None`)
`grab_release()`	Снимает с себя и дочерних виджетов поток событий приложения, взятый по `grab_set()`
`grab_set()`	Перехватывает на себя и дочерние виджеты поток событий приложения
`grab_set_global()`	То же, что и `grab_set()`, но принимает все события на дисплее. Используйте с осторожностью, т. к. другие приложения на том же дисплее не будут получать событий
`grab_status()`	Возвращает статус данного виджета в отношении перехвата событий: `None` (не перехватывает), `'local'` (перехватывает события данного приложения) или `'global'` (перехватывает события всего дисплея)

Следующие методы создают локальный цикл обработки событий для ожидания некоторых событий. Локальный цикл не мешает выполнению других событий.

Таблица 9.122. *Методы ожидания событий*

Метод	Описание
`wait_variable (name)`	Цикл ожидания присваивания переменной с именем `name`. Переменная `name` должна быть типа `IntVar`, `StringVar`, `DoubleVar` или `BooleanVar`

Таблица 9.122 *(окончание)*

Метод	Описание
`wait_window ([widget])`	Цикл ожидания уничтожения виджета `widget` (по умолчанию — себя)
`wait_visibility ([widget])`	Цикл ожидания изменения видимости виджета `widget` (то есть появления или сокрытия), по умолчанию `widget` — себя

Установка обработчиков событий. *Обработчики событий* — функции, вызываемые при наступлении определенных событий. Обработчику передается один аргумент — *дескриптор события (event descriptor)* (*см. статью "Описания событий" в разд. 9.19.2)* (там же вы узнаете о том, как задавать событие в виде строки `seq`). Для того чтобы обработчик вызывался при наступлении определенных событий, необходимо его привязать (bind) к событию и виджету (или классу виджетов). Методы для привязки (и отвязки) событий к обработчикам приведены в табл. 9.123.

Таблица 9.123. *Методы для привязки/отвязки обработчиков событий*

Метод	Описание
`bind(seq, handler[, "+"])`	Привязывает `handler` к обработке события `seq`. Если `"+"` не задан, старый обработчик отвязывается от обработки события в данном виджете
`unbind(seq)`	Отвязывает все обработчики события `seq` в данном виджете
`bind_all(seq, handler[, "+"])`	Привязывает `handler` к обработке события `seq` в рамках всего приложения. Если `"+"` не задан, старый обработчик отвязывается от обработки события в приложении
`unbind(seq)`	Отвязывает все обработчики события `seq` в приложении
`bind_class (cls, seq, handler [, "+"])`	Привязывает `handler` к обработке события `seq` для класса виджетов `cls`. Как и раньше, `"+"` служит для указания того, что новый обработчик добавляется, а не замещает старые
`unbind_class (cls, seq)`	Отвязывает все обработчики события `seq` для класса виджетов `cls`
`bindtags ([taglist])`	Возвращает (устанавливает) порядок поиска привязок для данного виджета. Кортеж `taglist` обычно состоит из имени самого виджета, класса виджета, имени окна верхнего уровня и специального имени, соответствующего приложению в целом

Работа с окном. Программа с графическим интерфейсом должна уметь управлять расположением показываемых окон. Методы, указанные в табл. 9.124, позволяют выстраивать окна ближе и дальше от переднего плана.

Таблица 9.124. *Методы для расположения окна*

Метод	Описание
`lift([wind])`, `tkraise([wind])`	Двигает окно на передний план. Если окно является дочерним, то оно оказывается перед своим родительским окном верхнего уровня. Если окно является окном верхнего уровня, оно становится перед всеми другими окнами на экране. Если задан параметр `wind`, окно становится перед указанным `wind` окном
`lower([wind])`	Аналогично `lift()`, но передвигает окно на задний план или за указанное окно `wind`

Получение информации о виджете. Методы, описанные в табл. 9.125, позволяют получать разнообразную информацию о виджете, об окне и экране, на котором находится виджет.

Таблица 9.125. *Методы, возвращающие информацию о виджете*

Метод	Описание
`winfo_cells() -> cells`	Число цветов в цветовой карте, которые использует данный виджет (от 2 до 255, почему-то даже для truecolor)
`winfo_children() -> list`	Список дочерних виджетов
`winfo_class() -> s`	Название класса виджета (например, `Button`)
`winfo_colormapfull() -> 0, 1`	Является ли цветовая карта полностью заполненной
`winfo_containing (x, y) -> w`	Виджет, который находится в позиции `(x, y)` относительно верхнего левого угла экрана, или `None`, если в данной координате нет виджетов, принадлежащих данному приложению
`winfo_depth() -> n`	Возвращает глубину цвета (в битах), используемую для вывода виджета. Обычно это 8, 16, 24 или 32 бита
`winfo_exists() -> 0, 1`	Существует ли окно виджета? (В норме должен возвращать `1`)
`winfo_pixels(dist) -> p`	Возвращает целочисленное расстояние в пикселах для заданного любым доступным для `Tkinter` расстояния `dist`

Таблица 9.125 *(окончание)*

Метод	Описание
`winfo_fpixels (distance) -> p`	То же, что и `winfo_pixels()`, но возвращает число с плавающей точкой
`winfo_geometry() -> g`	Возвращает расположение виджета на экране в виде строки `"WxH+X+Y"`, где `W` — ширина, `H` — высота, `X`, `Y` — координаты верхнего левого угла относительно родительского виджета или экрана
`winfo_width() -> W`	Ширина виджета в пикселах
`winfo_height() -> H`	Высота виджета в пикселах
`winfo_id() -> id`	Идентификатор виджета в оконной системе (зависит от используемой ОС)
`winfo_ismapped() -> 0, 1`	Возвращает истину, если виджет имеет область на экране
`winfo_manager() -> s`	Используемый для виджета менеджер расположения (`"pack"`, `"grid"` или `"place"`)
`winfo_name() -> s`	Имя виджета в Tk. Аналогично `str(self)`
`winfo_parent() -> s`	Имя родительского виджета в Tk или None
`winfo_reqheight() -> H`	Необходимая высота виджета. Минимальная высота, которая требуется для отображения виджета
`winfo_reqwidth() -> W`	Необходимая ширина виджета. Минимальная ширина, которая требуется для отображения виджета
`winfo_rootx(), winfo_rooty()`	Координаты виджета относительно левого верхнего угла экрана
`winfo_screen() -> s`	Возвращает имя экрана для виджета в виде строки `{":D.S"}`, где `D` — дисплей, а `S` — экран на этом дисплее. В операционных системах Windows и Macintosh всегда возвращается `":0.0"`
`winfo_screendepth() -> n`	Глубина цвета на экране, на котором выводится виджет
`winfo_screenwidth()`	Ширина экрана в пикселах
`winfo_screenheight()`	Высота экрана в пикселах
`winfo_screenmmwidth()`	Ширина экрана в миллиметрах
`winfo_ screenmmheight()`	Высота экрана в миллиметрах
`winfo_x(), winfo_y()`	Координаты левого верхнего угла виджета относительно левого верхнего угла родительского виджета

Примечание

Некоторые данные становятся доступными только после обработки менеджером окон.

Управление фокусом ввода с клавиатуры. При работе с приложениями графического интерфейса данные с клавиатуры могут поступать только в одно приложение и, как правило, они передаются виджету, имеющему *фокус ввода с клавиатуры*. Методы, описанные в табл. 9.126, позволяют управлять фокусом ввода.

Таблица 9.126. *Методы для манипулирования фокусом ввода с клавиатуры*

Метод	Описание
`focus_set()`, `focus()`	Принять на себя фокус ввода с клавиатуры данного приложения
`focus_force()`	Принять на себя фокус, даже если приложение не имеет фокуса в настоящий момент. Этот метод следует использовать с осторожностью. Кроме того, многих пользователей могут раздражать программы-выскочки, автоматически перехватывающие фокус ввода
`focus_get()`	Возвращает виджет, который принимает ввод с клавиатуры в данном приложении (или `{None}`, если такого нет)
`focus_displayof()`	Аналогично, для виджета, который имеет фокус на том же дисплее, где и данный виджет (или `None`, если такого нет)
`focus_lastfor()`	Возвращает виджет, который получит фокус, если окно с данным виджетом получит фокус от менеджера окон
`tk_focusNext()`	Возвращает виджет, получающий фокус вслед за данным (например, при нажатии клавиши табуляции <Tab>). Порядок обхода виджетов определяется их вложенностью друг в друга. Виджеты, у которых `takefocus` установлен в `0`, в обходе не участвуют. Обход происходит по циклу: от последнего виджета фокус передается первому
`tk_focusPrev()`	То же, но возвращает виджет, который принимает фокус перед данным

Работа с буфером обмена и выделением. Одним из незаменимых аксессуаров оконного интерфейса является возможность обмена информацией посредством специального *буфера обмена (clipboard)*. Рассматриваемые в табл. 9.127 методы позволяют управлять работой с выделенным текстом (selection) и буфером обмена.

Таблица 9.127. *Методы для работы с буфером обмена и выделением*

Метод	Описание
`clipboard_append(s)`	Добавляет строку `s` к буферу обмена
`clipboard_clear()`	Очищает буфер обмена
`selection_clear()`	Очищает выделение
`selection_get ([selection [, displayof]])`	Получает выделение. Именованный параметр `selection` позволяет задать имя выделения (в X Window по умолчанию `"PRIMARY"`), а `displayof` указывает виджет, который будет использован
`selection_handle (cmd)`	Устанавливает функцию `cmd` для обработки запросов на принадлежащие данному виджету выделения. Функция `cmd` должна принимать аргументы `OFFSET` (смещение от начала) и `LENGTH` (длина) и возвращать содержимое выделения длины `LENGTH` от позиции `OFFSET`. Также могут быть переданы именованные аргументы `selection` (имя выделения, по умолчанию `"PRIMARY"`) и `type` для типа выделения, например `"STRING"` (строка) или `"FILE_NAME"` (имя файла)
`selection_own()`	Стать владельцем выделения
`selection_own_get ([selection[, type]])`	Получить владельца выделения. Именованные аргументы `selection` и `type` задают имя выделения и его тип

Конфигурирование виджетов

Помимо методов, у объектов-виджетов есть свойства, которые можно задавать с помощью методов `config()` (`configure()` — его синоним), `cget()` и методов, подобных методам работы со словарями.

- `widget.config(option=value, ...)`
- `widget.configure(option=value, ...)`
- `widget.cget("option") -> value`
- `widget["option"] = value`
- `value = widget["option"]`
- `widget.keys()`

Метод `keys()` позволяет получить список опций данного виджета.

Замечание

Некоторые имена атрибутов и опций совпадают с ключевыми словами языка Python. По принятому в Python стилю написания программ, в этом случае после имени добавляется подчеркивание. Например, `class` будет задаваться как `class_`.

Стилевые опции

Каждый виджет имеет опции, касающиеся своего внешнего стиля: используемых *цветов*, шрифтов, *выравнивания текста*, оформления *границ*, формы курсора и т. п.

Каждый виджет имеет границу. Граница состоит из полосы для придания ощущения объема и полосы для обозначения того, что данный виджет выделен (находится в фокусе). В табл. 9.128 содержатся стилевые опции, относящиеся к границе виджета.

Таблица 9.128. *Опции, описывающие границу виджета*

Опция	Описание
`borderwidth` или `bd`	Ширина границы в пикселах
`relief`	Рельеф виджета: `SUNKEN` (утопленный), `RAISED` (приподнятый), `GROOVE` (окаймленный канавкой), `RIDGE` (обнесенный рубчиком) и `FLAT` (плоский)
`highlightthickness`	Ширина полосы выделения

В некоторых виджетах предусмотрен показ текста (в том числе, многострочного). В табл. 9.129 содержатся опции для задания свойств текста. (Некоторые виджеты предназначены специально для редактирования текста и потому имеют много дополнительных методов и опций.)

Таблица 9.129. *Опции оформления текста в виджете*

Опция	Описание
`anchor`	Управляет расположением содержимого внутри виджета (*anchor* в переводе с англ. — якорь). Возможные значения (константы определены в модуле `Tkinter`): `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW` и `CENTER`, см. рис. 9.4 с результатом работы программы в листинге 9.85
`font`	Шрифт
`justify`:	Выравнивание текста: `CENTER` (по центру), `LEFT` (по левому краю), `RIGHT` (по правому краю)
`padx`, `pady`	Зазор между содержимым виджета и его границами
`takefocus`	Опция, при истинном значении которой виджет может принимать фокус при нажатии пользователем клавиши <Tab>
`underline`	Опция для указания номера подчеркиваемого символа в тексте, содержащемся в виджете (или `-1`, если символ подчеркивать не нужно). Используется для обозначения "горячих" клавиш

Таблица 9.129 *(окончание)*

Опция	Описание
`width`, `height`	Ширина и высота виджета (в текстовых единицах измерения или в пикселах, в зависимости от типа содержимого)
`wraplength`	Максимальная ширина текста, выводимого без заворачивания строк

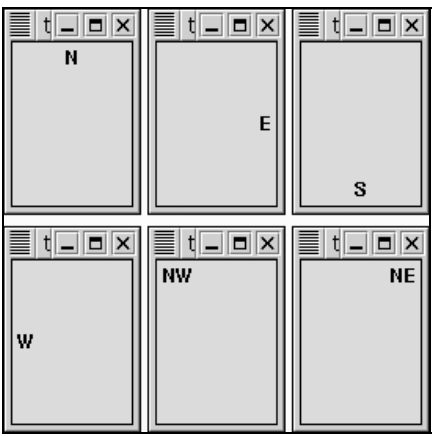


Рис. 9.4. Работа шести программ `tk_label_show.py`

Опция `font` требует отдельного объяснения.

Содержимое виджета

Виджеты обычно имеют некоторое наполнение: текст, картинка или другие виджеты. Есть виджеты, которые всецело посвящены содержанию других виджетов (например, `Frame`). Здесь же, в табл. 9.130, приведены опции для работы с текстовым или графическим содержимым виджета.

Таблица 9.130. *Опции для работы с содержимым виджета*

Опция	Описание
`text`	Строка с текстом. Строка может содержать признаки концов строк.
`bitmap`	Растровое изображение. Помимо предопределенных `"error"`, `"gray75"`, `"gray50"`, `"gray25"`, `"gray12"`, `"hourglass"`, `"info"`, `"questhead"`, `"question"` и `"warning"`, можно использовать файл в формате `xbm`, дав его название после `"@"`, например: `"@redhat.xbm"`. Если задана, текст не показывается

Таблица 9.130 *(окончание)*

Опция	Описание
`image`	Изображение как объект класса `BitmapImage`
`Textvariable`	Связывает переменную (типа `StringVar`) с текстовым содержимым виджета. При изменении переменной автоматически изменяется текст, содержащийся в виджете
`Variable`	Связывает переменную с состоянием виджета. При изменении переменной автоматически изменяется состояние виджета (и наоборот)

Задание шрифта. Модуль *tkFont*

Шрифт можно задать несколькими способами.

Замечание

Все сказанное ниже о шрифтах относится в основном к версии Tk не ниже 8.0.

Первый способ — указать кортеж из трех или менее элементов или, если имя семейства шрифт не содержит пробел, строку. В этих элементах задаются *семейство* шрифта, *размер (кегль)* шрифта и *начертание* шрифта:

```
(семейство шрифтов[, размер[, начертание]])
"семейство_шрифтов размер начертание"
```

где: `семейство_шрифтов` — общепринятое название шрифта вроде: `Helvetica`, `Times`, `Courier`, `Symbol`, `Verdana` и т. п.; `размер` — положительный — в пунктах, отрицательный — в пикселах; `начертание` — здесь может быть следующее: `bold` (полужирный), `italic` (курсив), `underline` (подчеркнутый), `overstrike` (зачеркнутый).

Следующий пример продемонстрирует различные сочетания свойств шрифтов.

Листинг 9.83. Файл show_fonts.py. Демонстрация свойств шрифтов

```
from Tkinter import *
root = Tk()
root.title("Шрифты")
f = Frame(root)
fonts = "Helvetica", "Times", "Courier", "Symbol", "Verdana"
decor = "", "bold", "underline", "overstrike"
sizes = "15", "-15", "20"
for fn in range(len(fonts)):
  k = 0
```

```
  for dc in decor:
    for sz in sizes:
      # формируем строку описания шрифта
      fnt = "%s\n%s %s" % (fonts[fn], sz, dc)
      b = Button(f, text=fnt, font=fnt, bg="white")
      b.grid(row=k, column=fn)
      k = k + 1

root.mainloop()
```

Результат работы этого примера показан на рис. 9.5. Используя листинг 9.83, вы можете экспериментировать с различными шрифтовыми оформлениями.

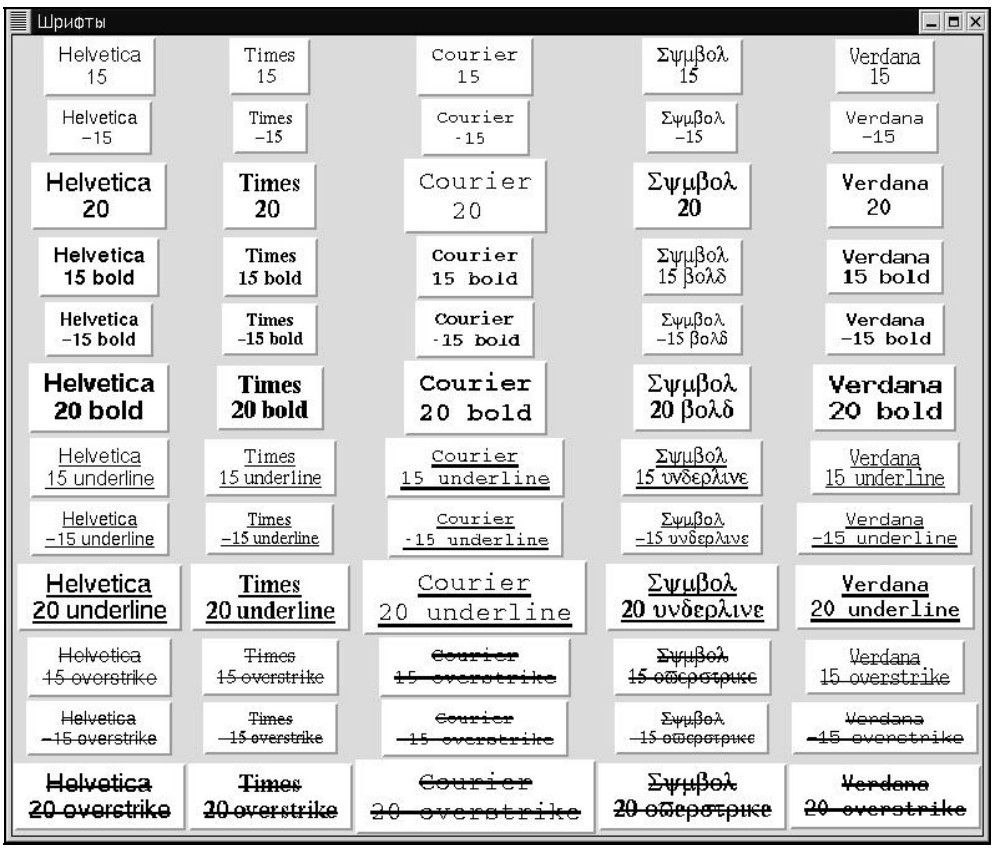


Рис. 9.5. Примеры шрифтов и сочетаний их свойств

Второй способ задания шрифтов — системные шрифты. В различных системах они называются по-разному, вот лишь некоторые названия: "fixed", "6x10", и т. п. (X Window); "ansi", "ansifixed", "device", "oemfixed", "system", "systemfixed" (Windows); "system" (Macintosh). Эти экранные шрифты нельзя комбинировать с размером и стилем. Их использования следует избегать из соображений переносимости программы.

Третий способ указания шрифтов имеет свои корни в системе X Window (обычная для UNIX графическая среда). Наверное, каждый, кто работал в X Window, вспомнит этот способ задания шрифта по следующему примеру:

```
-*-Courier-Bold-R-*--*-100-*-*-*-*-koi8-r
```

Четвертый способ задания шрифта использует модуль `tkFont`. Этот модуль предлагает класс `Font`, объекты которого имеют, среди прочего, методы для вычисления длины заданной строки в пикселах. В табл. 9.131 приведены опции объектов класса `Font`.

Таблица 9.131. *Опции объектов класса* `Font`

Опция	Описание
`Size`	Целое число, обозначающее размер (положительное — в пунктах, отрицательное — в пикселах)
`Weight`	Визуальная толщина. Возможные значения: `NORMAL` (нормальный), `BOLD` (полужирный)
`Slant`	Наклон. Возможные значения: `NORMAL` (нормальный), `ITALIC` (курсив)
`Underline`	Подчеркивание: 0 — нет, 1 — есть
`Overstrike`	Перечеркивание: 0 — нет, 1 — есть

С помощью метода `measure()` можно вычислить длину строки текста, как это сделано в следующем примере:

```
>>> import tkFont, Tkinter
>>> root = Tkinter.Tk()
>>> f = tkFont.Font(family="Arial", size=14, slant=tkFont.ITALIC)
>>> print f.measure("This string is this long:")
144
```

Цвета

Виджеты необязательно должны быть серыми. В табл. 9.132 показаны опции *цвета*, встречающиеся у виджетов. Не все опции из указанных присутствуют у всех виджетов, а только соответствующие смыслу виджета. Виджет бывает как в нормальном состоянии, так и выделен (когда он получает фокус), может

быть подсвечен (при прохождении курсора над виджетом). Кроме того, некоторые виджеты, например, кнопки, могут быть неактивными.

Таблица 9.132. *Цветовые опции виджетов*

Цветовая опция	Элемент виджета, которому соответствует опция
`activebackground`	Фон подсвеченного виджета
`activeforeground`	Текст подсвеченного виджета
`disabledforeground`	Текст неактивного виджета
`foreground`, `fg`	Текст виджета в обычном состоянии
`background`, `bg`	Фон виджета в обычном состоянии
`highlightcolor`	Полоса выделения в присутствии фокуса
`highlightbackground`	Полоса выделения при отсутствии фокуса

Цвета задаются как с помощью собственных имен: `"white"`, `"black"`, `"LightBlue"` и т. д., так и явным указанием компонентов основных цветов (красного, зеленого и синего) в шестнадцатеричном виде: `"#fff"`, `"#000000"`, `"#444444448888"`, где каждый из трех компонентов может быть 4, 8 или 16 битов. В следующем примере мы получаем подходящее для Tk представление цвета из трехэлементного кортежа:

```
>>> tk_color = "#%02X%02X%02X" % (200, 64, 125)
>>> print tk_color
#C8407D
```

Наверное, вы уже заметили некоторое сходство задания цвета в модуле `Tkinter` и в HTML.

Описания событий

Как уже говорилось, приложение с графическим интерфейсом как правило является событийно-ориентированным. В Tkinter для хранения информации о событии служат объекты класса `Event`. Они создаются им при возникновении событий. В табл. 9.133 приведены некоторые события, происходящие в среде графического интерфейса, которые `Tkinter` позволяет опознавать и обрабатывать.

Таблица 9.133. *Некоторые типы событий*

Обозначение	Событие
`KeyPress`	Нажатие клавиши на клавиатуре

Таблица 9.133 *(окончание)*

Обозначение	Событие
KeyRelease	Отпускание клавиши на клавиатуре
ButtonPress	Нажатие кнопки мыши
ButtonRelease	Отпускание кнопки мыши
Motion	Движение мыши (в рамках компетенции виджета)
Enter	Вхождение курсора в пределы виджета
Leave	Выход курсора за пределы мыши
MouseWheel	Кручение колесика мыши
Visibility	Изменение видимости окна
FocusIn	Окно получает фокус
FocusOut	Окно теряет фокус
Reparent	Изменение родителя окна
Destroy	Уничтожение окна
Activate	Окно становится активным
Deactivate	Окно становится неактивным

Описание события (например, при привязке событий к виджетам) строится по следующей схеме (часть элементов может быть опущена):

`<модификаторы-тип-детализация>`

где `модификаторы` — указание одного или нескольких сопутствующих обстоятельств и уточнений, например состояние клавиши <Shift> или вид щелчка мыши (двойной, тройной); `тип` — тип события (табл. 9.133); `детализация` — дополнительная информация о событии, например, номер кнопки мыши.

Например, "`<ButtonPress-3>`" (нажатие правой кнопки мыши), "`<3>`" (то же), "`<Double-Button-1>`" (двойной щелчок левой кнопки мыши), "`<Shift-Double-Button-1>`" (двойной щелчок левой кнопки мыши при нажатой клавише <Shift>), "`<space>`" (нажатие клавиши <пробел>), "`y`" (нажатие клавиши <Y>) и т. п.

Практически все клавиши на стандартной клавиатуре персонального компьютера могут быть применены в описании события. Вот их названия (их следует помещать в <>):

`Cancel, BackSpace, Tab, Return, Shift_L, Control_L, Alt_L,`

`Pause, Caps_Lock, Escape, Prior, Next, End, Home, Left,`

Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num_Lock, Scroll_Lock, space, less.

Примечание

Prior соответствует клавише <PageUp>, Next — <PageDown>, а space и less — <пробел> и "<" соответственно.

Объект класса Event имеет несколько атрибутов. В табл. 9.134 атрибуты событий приведены вместе с описаниями и указанием событий.

Таблица 9.134. *Атрибуты событий*

Атрибут	Значение	События, имеющие данный атрибут
serial	Серийный номер события	
num	Номер кнопки мыши	ButtonPress, ButtonRelease
focus	Имеет ли окно фокус	Enter, Leave
height	Высота окна	Configure, Expose
width	Ширина окна	Configure, Expose
keycode	Код клавиши	KeyPress, KeyRelease
state	Состояние события в виде целого числа или строки	ButtonPress, ButtonRelease, Enter, KeyPress, KeyRelease, Leave, Motion, Visibility
time	Время возникновения события	
x	X-координата мыши	
y	Y-координата мыши	
x_root	X-координата мыши на экране	ButtonPress, ButtonRelease, KeyPress, KeyRelease, Motion
y_root	Y-координата мыши на экране	ButtonPress, ButtonRelease, KeyPress, KeyRelease, Motion
char	Символ	KeyPress, KeyRelease
keysym	Символ	KeyPress, KeyRelease
keysym_num	Символ в виде числа	KeyPress, KeyRelease
type	Тип события в виде номера	
widget	Виджет, в котором случилось событие	
delta	Приращение движения колесика мыши (например, IntelliMouse)	MouseWheel

Листинг 9.84 позволяет проследить атрибуты события в текстовом окне. Для примера используется событие `Motion`.

Листинг 9.84. Файл tk_events.py. Примеры обработки событий в `Tkinter`

```
from Tkinter import *
tk = Tk()
txt = Text(tk)
txt.pack()

def give_info(event):
  txt.delete("1.0", END)
  items = event.__dict__.items()
  items.sort()
  for k, v in items:
    txt.insert(END, "%s: %s\n" % (k, v))

txt.bind("<Motion>", give_info)
tk.mainloop()
```

9.19.3. Различные виджеты

Далее мы рассмотрим некоторые методы наиболее типичных классов виджетов. Так как каждый виджет имеет более 150 методов, то описать все из них в рамках данной книги не представляется возможным. Заинтересованные читатели могут обратиться к оригинальной документации по модулю `Tkinter` или языку Tcl/Tk или изучить виджеты, пользуясь диалоговым режимом, как это описано в *разд. 7.6*.

Мы же рассмотрим некоторые наиболее типичные виджеты. Почти каждый из них имеет некоторые особенности по сравнению с "базовым виджетом", дополнительные методы и опции.

Окна верхнего уровня

Окна верхнего уровня — виджеты, которые определяют самостоятельные окна. (Строго говоря, класс `Toplevel`, порождающий окна верхнего уровня, не является подклассом класса `Widget`, но нам удобнее называть окна верхнего уровня виджетами). Окна обычно управляются менеджером окон, действующим в данной системе. Для X Window написано много разнообразных менеджеров окон, тогда как для MacOS и Windows выбора практически нет: нужно использовать стандартный менеджер окон этих ОС.

В табл. 9.135 приведены группы методов, присущие окнам.

Таблица 9.135. *Основные методы виджетов окон верхнего уровня*

Метод	Описание
`group(window)`	Добавляет окно в группу окон, контролируемых окном `window`. Окна группы убираются и появляются вместе с управляющим окном
`overrideredirect([flag])`	Получить или установить значение флага. Если флаг не ноль, границы окна и полоска с названием не будут прорисованы. Такое окно нельзя будет изменять в размерах и двигать обычными средствами
`protocol(name, func)`	Регистрирует функцию `func`, которая будет вызываться в ответ на сообщение `name` менеджера окон. Сообщения менеджера окон: `WM_DELETE_WINDOW` (окно удаляется), `WM_TAKE_FOCUS` (окно получило фокус), `WM_SAVE_YOURSELF` ("сохраняйся!")
`title([s])`	Получает или устанавливает заголовок окна
`transient([master])`	Придает окно виджету `master` (по умолчанию — родительскому виджету). Такие окна всегда рисуются поверх `master`, минимизируются и восстанавливаются вместе с ним
`deiconify()`	Показывает ранее минимизированное окно
`iconify()`	Минимизирует окно, убирая его в иконку или панель задач
`state()`	Показывает состояние окна: `'iconic'` (минимизированное), `'withdrawn` (убранное), `'normal'` (нормальное) и `'icon'` (иконка)
`withdraw()`	Убирает окно с экрана (но не закрывает)
`aspect([a, b, c, d])`	Устанавливает пропорции окна, сохраняя отношение "ширина к высоте" в промежутке между `a/b` и `c/d`. Если аргументы не заданы, метод возвращает текущее значение пропорции в виде кортежа из четырех значений
`geometry([newGeometry])`	Получает или устанавливает расположение окна на экране в виде строки следующего формата: `WxH+X+Y`, где `W` — ширина, `H` — высота, `X`, `Y` — координаты верхнего левого угла. Единица измерения — пиксел (если окно не использует привязку к узлам сетки: тогда единица может быть иной)
`maxsize([W, H])`	Получает или устанавливает максимальный размер окна, ширина `W`, высота `H`
`minsize([W, H])`	Получает или устанавливает минимальный размер окна, ширина `W`, высота `H`

Таблица 9.135 *(окончание)*

Метод	Описание
`resizable ([hor, vert])`	Устанавливает флаги возможности изменения размеров окна по горизонтали (`hor` равен 1) и вертикали (`vert` равен 1). Если задан без аргументов, возвращает строку формата `"%s %s" % (hor, vert)`

Сразу после создания окно принимает состояние `'normal'`. Внутри окон располагаются другие виджеты. Ниже мы рассмотрим некоторые из них.

Этикетка (виджет *Label*)

Этот тип виджета используется для показа текста или графического изображения (bitmap). *Этикетка* — достаточно стандартный и часто используемый виджет, поэтому он не имеет дополнительных опций к рассмотренным в *разд. 9.19.2*.

В листинге 9.85 используется смена якоря и текста этикетки для наглядной демонстрации названий сторон света. В результате выполнения примера появится рисунок, на котором видны шесть одновременно работающих программ `tk_label_show.py` в разных состояниях.

Листинг 9.85. Файл tk_label_show.py. Демонстрация этикеток

```
import time
from Tkinter import Tk, Frame, Label, N, NE, E, SE, S, SW, W, NW, CENTER
tk = Tk()
f = Frame(tk)
f.pack()
l = Label(f, anchor=CENTER, text="CENTER", width=10, height=7)
l.pack()
time.sleep(1)
for i in range(5):
  for a in ["N", "NE", "E", "SE", "S", "SW", "W", "NW"]:
    l.config(anchor=eval(a), text=a)
    l.update_idletasks()  # для перерисовки изображения
    time.sleep(1)
```

Текстом этикетки можно управлять и с помощью переменной типа `StringVar`:

```
from Tkinter import *
```

```
tk = Tk()
tv = StringVar()
Label(tk, textvariable=tv, relief="groove").pack()
tv.set("Этикетка")
tk.mainloop()
```

Кнопка (виджет *Button*)

Виджет кнопка (`Button`) служит для предоставления пользователю возможности выполнять определенное действие, соответствующее приведенному на кнопке тексту или пиктограмме. Например:

```
import Tkinter
tk = Tkinter.Tk()
b = Tkinter.Button(tk, text="Exit", command=tk.destroy)
b.pack()
tk.mainloop()
```

В этом примере кнопка **Exit** служит для выхода из программы по закрытии окна `tk`.

Кнопки имеют дополнительные методы: `flash()` для того, чтобы заставить кнопку "поморгать", и `invoke()` для выполнения связанной с кнопкой команды.

Кнопки также имеют специфичные для них опции. Эти опции представлены в табл. 9.136.

Таблица 9.136. *Опции, специфичные для кнопок*

Опция	Описание
`command`	Функция, метод или любой объект, который будет вызываться в случае нажатия пользователем на кнопку или вызове метода `invoke()`
`default`	Указание того, что кнопка является кнопкой по умолчанию в данном диалоговом окне. Такие кнопки могут выделяться особым образом в оконном интерфейсе
`state`	Состояние кнопки. Возможные значения: `NORMAL`, `ACTIVE` и `DISABLED` (константы из модуля `Tkinter`). От состояния кнопки зависит ее представление на экране.

В листинге 9.86 приведен пример использования кнопок в приложении "Калькулятор".

Листинг 9.86. Файл calctk.py. Калькулятор

```
from Tkinter import *

global display_var
buttons = [["7", "8", "9", "(", ")",],
           ["4", "5", "6", "+", "-",],
           ["1", "2", "3", "*", "/",],
           ["0", ".","**", "C", "=",],]
def key(key):
  """Функция возвращает функцию для обработки нажатия на клавишу"""
  def put_char(char=key):
    """Обработка нажатия на клавишу калькулятора"""
    global display_var
    display_var.set(display_var.get()+char)
  return put_char

def evalResult():
  """Обработка нажатия на клавишу [=]"""
  global display_var
  try:
    display_var.set(eval(display_var.get()))
  except:
    display_var.set("ERROR")

def clearDisplay():
  """Обработка нажатия на клавишу [C]"""
  global display_var
  display_var.set("")

tk = Tk()
display_var = StringVar("")  # переменная - состояние табло
display = Label(tk, textvariable=display_var,
                bg="#008000", fg="#FFFFFF", relief=RAISED)
display.pack(anchor=NW, fill=BOTH)  # прикручиваем табло
display.configure(justify=LEFT)
kbd = Frame(tk)
kbd.pack(anchor=NW)  # прикручиваем панель с кнопками
for row in range(len(buttons)):
```

```
  for col in range(len(buttons[row])):
    btn_text = buttons[row][col]
    if btn_text == "=":
      cmd = evalResult
    elif btn_text == "C":
      cmd = clearDisplay
    else:
      cmd = key(btn_text)
    # создаем кнопку с текстом btn_text, которая
    # выполняет команду cmd
    btn = Button(kbd, text=btn_text, command=cmd)
    # помещаем кнопку в соответствующее место на клавиатуре
    btn.grid(row=row, col=col, sticky=W+E+N+S)
tk.mainloop()
```

На рисунке 9.6 показан "Калькулятор" в процессе работы.



Рис. 9.6. Приложение "Калькулятор"

Переключатель (*Radiobutton*) и флажок (*Checkbutton*)

Для ввода данных в GUI-программах часто применяются *переключатели (виджет `Radiobutton`)* и *флажки (виджет `Checkbutton`)*. В табл. 9.137 представлены методы, специфичные для этих видов виджетов. Серия переключателей используется для задания одного из нескольких альтернативных значений, а флажок используется для задания значения включено-выключено.

Таблица 9.137. *Методы переключателей и флажков*

Метод	Описание
`deselect()`	Убирает выбор кнопки
`flash()`	Заставляет "поморгать" кнопкой

Таблица 9.137 *(окончание)*

Метод	Описание
`invoke()`	Выполняет связанную с кнопкой команду
`select()`	Выбирает кнопку
`toggle()`	Изменяет состояние флажка на обратное (для флажка)

Переключатели и флажки имеют те же опции, что и обычные кнопки, а также дополнительные опции, приведенные в табл. 9.138.

Таблица 9.138. *Опции переключателей и флажков*

Опция	Описание
`indicatoron`	Управляет индикатором состояния. Если равна `1`, кнопка имеет отдельный индикатор, если `0` — состояние индицируется вдавленностью кнопки
`selectcolor`	Цвет индикатора
`selectimage`	Графическое изображение для индикатора
`offvalue`, `onvalue`	Значения флажка для выключенного и включенного состояний

В листинге 9.87 приведен пример использования переключателей и флажков. На экране появляются три переключателя и два флажка, а чуть ниже три этикетки, которые отражают состояния этих кнопок. Заметьте, что на группу переключателей приходится одна переменная, тогда как каждый флажок требует своей переменной (иначе флажки будут показывать одно состояние).

Листинг 9.87. Файл cbut.py. Пример с переключателем и флажками

```
from Tkinter import *

tk = Tk()
f = Frame(tk)
f.pack()

radio_btns = "1", "2", "3"
radio_var = StringVar()
chk_btns = ("A", StringVar()), ("B", StringVar())
```

```
# Три переключателя:
for btn in radio_btns:
  Radiobutton(f, text=btn, variable=radio_var, value=btn).pack()
# Два флажка:
for btn, var in chk_btns:
  Checkbutton(f, text=btn, variable=var,
              onvalue=btn, offvalue="").pack()
# Этикетки для индикации состояний:
Label(f, textvariable=radio_var).pack()
Label(f, textvariable=chk_btns[0][1]).pack()
Label(f, textvariable=chk_btns[1][1]).pack()

tk.mainloop()
```

На рис. 9.7 приведено состояние программы листинга 9.87 после выбора кнопки **3** и флажка **A**.



Рис. 9.7. Программа `cbut.py` в работе

Поле ввода (виджет *Entry*)

Виджет поле ввода (`Entry`) используется для ввода/вывода строки текста. Этот виджет имеет много дополнительных методов (табл. 9.139) и некоторые дополнительные опции (табл. 9.140).

Виджет `Entry` имеет *точку вставки*, которую пользователь видит как мигающий *I-курсор*, а также возможность выделять текст. Для обозначения различных мест в редактируемом тексте можно использовать:

- константы `0` (начало текста), `END` (конец текста), `ANCHOR` (начало выделения), `INSERT` (позиция точки вставки);
- числовые индексы (по аналогии с индексами в строках языка Python);
- величину смещения курсора мыши от левого края (задается как `"@x"`, где `x` — требуемое смещение в пикселах).

Таблица 9.139. *Методы виджета* `Entry`

Метод	Описание
`insert(index, text)`	Вставляет текст `text` по индексу `index`
`delete(index)` или `delete(index1, index2)`	Удаляет символ с индексом `index` или символы от `index1` до `index2` (аналогично `del s[index]` и `del s[index1:index2]` для строк)
`icursor(index)`	Устанавливает точку вставки по индексу `index`
`get() -> s`	Возвращает текст в поле ввода
`index(index) -> index1`	Возвращает числовой индекс `index1`, соответствующий индексу `index`
`select_adjust(index)`	Расширяет выделение до указанного в `index` места
`select_clear()`	Убирает выделение
`select_from(index)`	Начинает новое выделение с индекса `index`
`select_to(index)`	Выделяет текст от `ANCHOR` до `index`
`select_range(start, end)`	Выделяет текст от `start` до `end`
`select_present() -> 0, 1`	Флаг, индицирующий присутствие выделения в тексте

Примечание

Методы `select_` имеют синонимы с `selection_`.

Кроме того, виджет `Entry` поддерживает методы для прокрутки: `scan_mark()`, `scan_dragto()`, `xview()`, `xview_moveto()` и `xview_scroll` *(см. статью "Текст (виджет Text)" в этом разделе).*

Таблица 9.140. *Опции виджета* `Entry`

Опция	Описание
`exportselection`	Флаг, при истинном значении которого выделение автоматически попадает в буфер обмена (в системе X Window это поведение является нормой, т. к. для копирования текста из одного приложения в другое достаточно выделить текст в одном приложении и нажать среднюю кнопку мыши в том месте, куда нужно вставить выделение)
`insertbackground`	Цвет I-курсора
`insertofftime`, `insertontime`	Продолжительности фаз мерцания курсора: `insertofftime` — продолжительность выключения, `insertontime` — включения, в миллисекундах

Таблица 9.140 *(окончание)*

Опция	Описание
insertwidth	Ширина I-курсора
selectbackground	Цвет фона выделения
selectborderwidth	Ширина границы выделения
selectforeground	Цвет текста выделения
Show	Если эта строка не пустая, вместо символов пользователю показываются символы этой строки. Например, "*" вместо символов вводимого пароля
xscrollcommand	Указывает функцию, которая будет вызываться в случае горизонтальной прокрутки поля ввода

В следующем примере по мере ввода в поле символов изменяется список файлов, начинающихся на эти символы (см. листинг 9.88 и результат на рис. 9.8).

Листинг 9.88. Файл incrfile.py. Инкрементный подбор имени файла

```
from Tkinter import *

def list_files(x, y, z):
  """Приводит список файлов, начинающихся на введенные символы"""
  import glob
  lst = glob.glob(filena.get()+"*")  # получаем список
  files.delete(0, END)  # очищаем старый список
  for file in lst:
    files.insert(END, file)  # вводим файлы в список

tk = Tk()
f = Frame(tk)
f.pack()
filena = StringVar()  # переменная со значением поля ввода
filena.trace("w", list_files)  # отслеживаем записи в нее
filename_entry = Entry(f, textvariable=filena)
filename_entry.pack()
files = Listbox(f)
files.pack()
tk.mainloop()
```



Рис. 9.8. Программа `incrfile.py` после ввода символов `mk`

Меню

Виджет меню (`Menu`) служит для организации меню: *меню верхнего уровня*, *всплывающих (`popup`)* и *ниспадающих (`pulldown`) меню* (рис. 9.9). Tk и, соответственно, `Tkinter` по умолчанию предоставляют возможность отрывных (tearoff) меню. При отрыве меню появляется маленькое окошечко с пунктами оторванного меню, что позволяет пользователю удобнее организовать рабочее пространство. Там, где это необходимо, меню могут быть вложенными, *каскадными (cascade)*.

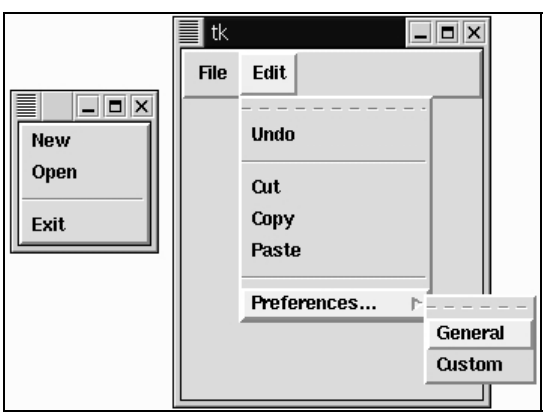


Рис. 9.9. Различные виды меню

Следующий пример (листинг 9.89) был использован для построения меню, показанного на рис. 9.9. Меню **File** было оторвано от основного меню и положено в виде отдельного окна слева от основного окна приложения. Затем было выбрано меню **Edit-Preferences...**, что и запечатлено на снимке экрана.

Листинг 9.89. Файл menu_show.py. Различные виды меню

```
from Tkinter import *

def show(msg="menu"):
  def print_msg(message=msg):
    print message
  return print_msg

tk = Tk()
main_menu = Menu(tk)
tk.config(menu=main_menu)
file_menu = Menu(main_menu)
main_menu.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="New", command=show("New"))
file_menu.add_command(label="Open", command=show("Open"))
file_menu.add_separator()
file_menu.add_command(label="Exit", command=tk.destroy)
edit_menu = Menu(main_menu)
main_menu.add_cascade(label="Edit", menu=edit_menu)
edit_menu.add_command(label="Undo", command=show("Undo"))
edit_menu.add_separator()
edit_menu.add_command(label="Cut", command=show("Cut"))
edit_menu.add_command(label="Copy", command=show("Copy"))
edit_menu.add_command(label="Paste", command=show("Paste"))
edit_menu.add_separator()
pref_menu = Menu(edit_menu)
pref_menu.add_command(label="General", command=show("General"))
pref_menu.add_command(label="Custom", command=show("Custom"))
edit_menu.add_cascade(label="Preferences...", menu=pref_menu)
tk.mainloop()
```

Текст (виджет *Text*)

Текстовый виджет (`Text`) предоставляет возможности показа, редактирования и форматирования текста. Помимо собственно знаков текст может содержать маркеры (marks), встроенные окна с произвольными виджетами, изображения. Для простоты мы будем называть все объекты, встречающиеся в тексте, символами. Для указания позиций в тексте используются индексы, которые аналогично индексам в Python соответствуют межзнаковым интервалам. В табл. 9.141 приведены возможные варианты указания индекса для

виджета Text. С помощью метода index() можно преобразовать любой вид индекса к виду "строка-колонка".

Таблица 9.141. *Индексы в текстовом виджете*

Индекс	Формат	Описание
Строка-колонка	"строка.колонка"	Номера строк начинаются с единицы, а колонок — с нуля. Вместо номера колонки можно задать end, что будет соответствовать символу конца строки
Точка вставки	INSERT	Соответствует точке вставки
Символ под курсором	CURRENT	Соответствует символу, ближайшему к курсору мыши
Конец текста	END	Позиция сразу после последнего символа в тексте
Пометки (теги)	"тег.first", "тег.last"	Индексы начала и конца помеченного тег участка текста. Для выделения предопределена пометка sel. Пользователь может определить свои пометки с помощью соответствующих методов
Маркеры	Имя	Пользователь может определять свои именованные позиции в тексте (аналогично INSERT, CURRENT) с помощью соответствующих методов. При редактировании текста маркеры будут сдвигаться в соответствии с заданными для них правилами
Выделение	SEL_FIRST, SEL_LAST	Соответствуют началу и концу выделенного текста. Если текст не выделен, вызывается исключение TclError
Координаты в окне	"@x,y"	Соответствует символу, ближайшему к точке с координатами X, Y
Встроенные в текст объекты	Объект	Могут служить индексами для указания в тексте своих позиций
Индекс с модификатором	str(индекс)+"модификатор"	Служат для указания позиции относительно индекса. В качестве модификатора могут быть заданы строки следующего формата: "+N chars" (вперед на N символов) , "-N chars" (назад на N символов), "+N lines" (вперед на N строк), "-N lines (назад на N строк), "linestart" (к началу строки), "lineend" (к концу строки), "wordstart" (к началу слова), "wordend" (к концу слова). Слова могут быть сокращены, если это не вызывает неопределенности. Например, "+ 1 chars" и "+1c" — одно и то же

В табл. 9.142 содержатся некоторые методы, которые имеет виджет Text. Некоторые методы в табл. описаны обобщенно, сразу для многих объектов. Вместо объект можно подставить mark (маркер), window (окно), image (изображение) или tag (пометка). Под опциями понимаются заданные именованные аргументы, соответствующие опциям. Возможные опции даны вместе с описанием метода. Вместо указания индексы можно использовать либо один параметр, который является индексом конкретного символа, либо два параметра — индексы для указания части текста от первого (включительно) до второго индекса (исключительно).

Таблица 9.142. *Некоторые методы виджета* Text

Метод	Описание
bbox(index)- > (x, y, width, height)	Возвращает координаты символа и его ширину и высоту. Может возвратить None, если символ не виден на экране
compare(index1, op, index2) -> 0, 1	Возвращает результат сравнения двух позиций в тексте. Параметр op может принимать значения "<", "<=", "==", "!=", ">=", ">"
delete(индексы)	Удаляет символ или символы
dlineinfo(index) -> (x, y, width, height, offset)	Возвращает координаты строки текста, содержащей указанный index символ, или None, если строка не видна на экране. В пятиэлементном кортеже возвращаются координаты строки в окне, ее ширины и высоты и расстояние от верхней границы до базисной линии
get(индексы) -> s	Возвращает символ или символы
image_create(index, опции)	Вставляет изображение в указанную index позицию. Могут применяться следующие опции: align (выравнивание) может быть TOP (по верхней границе), CENTER (по центру), BOTTOM (по нижней границе) или BASELINE (по базисной линии текста); image — объект с изображением; name — имя для изображения (по умолчанию модуль Tkinter дает свое имя); padx, pady задают величину окружающего окно пустого пространства
index(index) -> index1	Приводит index к виду "строка.колонка"
insert(index, text [, tags])	Вставляет текст в заданную позицию index (например, INSERT или END) текст text, отмечая этот текст пометками tags
mark_gravity(mark)	Возвращает значение предпочтений маркера

Таблица 9.142 *(продолжение)*

Метод	Описание
`mark_gravity(mark, keepside)`	Устанавливает предпочтения маркера в случае вставки текста в его позицию. Параметр `keepside` может принимать значения `LEFT` (держаться левой стороны) или `RIGHT` (правой, это значение по умолчанию)
`mark_set(mark, index)`	Устанавливает маркер с именем `mark` в позицию `index`. Если маркер не существовал, он создается
`mark_unset(mark)`	Удаляет маркер `mark`
`search(pat, index, опции)`	Ищет строку по образцу `pat`. Опции следующие: `forwards` (вперед) или `backwards` (назад) — ненулевое значение одной из этих опций указывает направление; опции `exact` (точное совпадение) или `regexp` (регулярное выражение) — способ сопоставления; `nocase` — включает регистронезависимый поиск; `stopindex` — задает индекс, до которого искать; `count` — задает переменную (`IntVar`), в которой будет возвращена длина найденного фрагмента
`see(index)`, `yview(index)`	Изменяет прокрутку таким образом, чтобы символ по индексу `index` стал виден
`tag_add(tag, индексы)`	Помечает текст, заданный `индексами`, пометкой `tag`
`tag_bind(tag, seq, func[, "+"])`	Привязывает событие `seq` к тексту, помеченному `tag`. При наступлении соответствующего события вызывается `func`. *(См. статью "Описания событий" в разд. 9.19.2.)*
`tag_lower(tag[, below])`	Помещает пометку `tag` в самый низ списка пометок (или ниже, чем пометка `below`, если она задана). Если участок текста имеет несколько пометок, пометки, находящиеся выше по списку, имеют больший приоритет. То есть метод уменьшает приоритет пометки
`tag_nextrange(tag, индексы -> (start, end))`	Находит следующий участок текста, помеченный `tag`, возвращая индексы начала и окончания этого участка. Если указанный в `индексах` символ является началом участка, помеченного `tag`, то возвращаются индексы, соответствующие этому участку. Если следующего участка не найдено, возвращается пустой кортеж `()`
`tag_prevrange(tag, индексы -> (start, end))`	Находит предыдущий участок текста, помеченный `tag`, возвращая индексы начала и окончания этого участка. Если заданный `индексами` символ помечен `tag`, метод возвращает данные для участка, в котором находится этот символ, если это не первый символ помеченного участка

Таблица 9.142 *(окончание)*

Метод	Описание
`tag_raise(tag[, above])`	Помещает пометку `tag` на самый верх списка пометок (или выше, чем пометка `above`). (См. описание метода `tag_lower()`.)
`tag_ranges()`	Возвращает кортеж вида `(start1, stop1, start2, stop2, ...)` с индексами начал и концов всех участков, помеченных `tag`
`tag_remove(tag, индексы)`	Снимает пометку `tag`, заданную `индексами`
`tag_unbind(tag, seq)`	Отвязывает событие `seq` от текста, помеченного `tag`
`window_create(index, опции)`	Вставляет виджет в указанную `index` позицию. Могут применяться следующие опции: `align` (выравнивание) может быть `TOP` (по верхней границе), `CENTER` (по центру), `BOTTOM` (по нижней границе) или `BASELINE` (по базисной линии текста); `create` — объект для вызова при первом показе окна (должен создать виджет, дочерний для текстового виджета, и возвратить его); `padx`, `pady` задают величину окружающего окно пустого пространства; `stretch` (растяжка на всю длину строки), которая может быть включена (`ON`) или выключена (`OFF`); `window` (окно) — виджет для вставки в текст
`xview(), yview()`	Возвращает кортеж из двух значений, лежащих в диапазоне от `0` до `1` (включительно), обозначающий долю видимой в виджете части текста
`xview(MOVETO, xoff), yview(MOVETO, yoff)`	Корректирует прокрутку в текстовом виджете так, чтобы левый (верхний) обрез текста соответствовал позиции `xoff` (`yoff`) — числу от `0` до `1`, обозначающему долю текста, лежащего левее (выше) видимого в окне текста
`xview(SCROLL, step, what), yview(SCROLL, step, what)`	Прокручивает текст на `step` единиц. Параметр `what` может принимать значения `UNITS` (строки, колонки) или `PAGES` (страницы)
`объект_cget(index, option)`	Возвращает значение опции `option`
`объект_config(index, опции)`	Устанавливает опции `объекта`. Синоним `объект_configure()`
`объект_names()`	Возвращает имена всех `объектов`, использованных в тексте

Примечание

Базисной линией (baseline) в полиграфии называется линия, соответствующая нижней границе символов вроде "m". Некоторые символы могут иметь "хвостики", опускающиеся ниже базисной линии (как, например у "p").

В листинге 9.90 показано построение окна для редактирования текста, использующего вертикальную и горизонтальную полосы прокрутки.

Листинг 9.90. Файл txtscroll.py. Пример текстового поля с прокруткой

```
from Tkinter import *
import sys
tk = Tk()               # создаем окно верхнего уровня,
txtframe = Frame(tk)  # фрейм с текстом и полосами прокрутки
txtframe.pack()
txt = Text(txtframe)  # текстовый виджет
txt.grid(row=0, column=0)
                        # горизонтальная полоса прокрутки
hor_bar = Scrollbar(txtframe, command=txt.xview, orient=HORIZONTAL)
hor_bar.grid(row=1, column=0, sticky=W+E+N+S)
                        # вертикальная полоса прокрутки
vert_bar = Scrollbar(txtframe, orient=VERTICAL, command=txt.yview)
vert_bar.grid(row=0, column=1, sticky=W+E+N+S)
                        # настраиваем обратную связь от текста
                        # к полосам прокрутки
txt.config(yscrollcommand=vert_bar.set, xscrollcommand=hor_bar.set,
           wrap=NONE)

for x in range(1, 1000):  # наполняем текстовый виджет содержимым
  txt.insert(END, str(2L**x)+"\n")

tk.mainloop()           # запускаем цикл событий
```

На рис. 9.10 показан результат работы программы `txtscroll.py`.

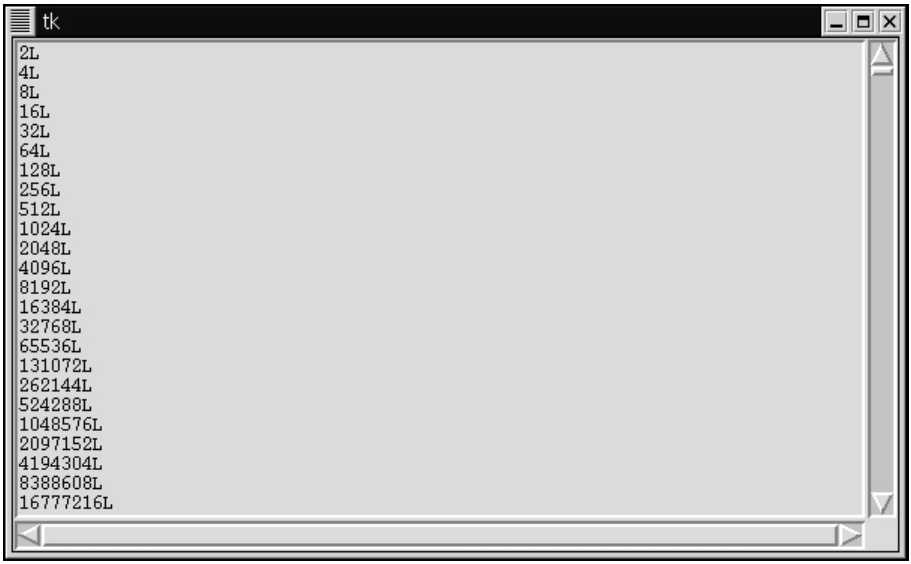


Рис. 9.10. Результат работы программы `txtscroll.py`

Пометкам (тегам) в тексте соответствуют специальные объекты, которые имеют свой набор методов, позволяющих придавать охваченному пометкой тексту нужные свойства (табл. 9.143).

Таблица 9.143. *Опции пометок (тегов)*

Опция	Описание
`background`	Цвет фона
`bgstipple`	Битовая маска для фона (например, `"gray50"` или другой `BitmapImage`)
`borderwidth`	Ширина границы текста
`fgstipple`	Битовая маска для изображения текста
`font`	Шрифт
`foreground`	Цвет текста
`justify`	Выравнивание строки. Может быть: `LEFT` (по левому краю), `CENTER` (по центру) или `RIGHT` (по правому краю). Определяется по первому символу в строке
`lmargin1`	Левая граница первой строки блока текста, имеющего эту пометку
`lmargin2`	Левая граница всех строк блока текста, имеющего эту пометку, кроме первой

Таблица 9.143 *(окончание)*

Опция	Описание
`offset`	Смещение текста относительно базисной линии. Положительные значения соответствуют смещению вверх. Опция может быть использована для реализации верхних и нижних индексов и т. п.
`overstrike`	Если не ноль, применяется перечеркивание текста
`relief`	Рельеф текста. См. табл. 9.128. По умолчанию `FLAT`
`rmargin`	Правая граница строк блока текста
`spacing1`, `spacing2`, `spacing3`	Величина отбивки перед первой строкой, внутри и после последней строки блока текста соответственно
`tabs`	Задает последовательные значения позиций табуляции (в виде кортежа значений), либо одно значение, задающее одинаковую величину табуляции
`underline`	Если не ноль, подчеркивание текста
`wrap`	Указывает способ заворачивания длинных строк текста. Может быть `NONE` (не заворачивать), `CHAR` (по символам) или `WORD` (по словам). По умолчанию `None` (как обычно, эти три константы определены в модуле `Tkinter`)

В табл. 9.144 содержатся опции виджета `Text`. Кроме того, в виджете действуют почти все опции виджета `Entry` *(см.* табл. 9.140).

Таблица 9.144. *Опции виджета* `Text`

Опция	Описание
`setgrid`	Если "истина", `Tkinter` пытается изменять размер окна с текстом с шагом ширины символа (чтобы символы были видны целиком)
`xscrollcommand`	Указывает функцию, которая будет вызываться в случае горизонтальной прокрутки поля ввода (применяется для связи с полосой прокрутки)
`yscrollcommand`	Указывает функцию, которая будет вызываться в случае вертикальной прокрутки поля ввода (применяется для связи с полосой прокрутки)

Кроме того, для всего текста можно задать опции пометок *(см.* табл. 9.143): `background`, `borderwidth`, `font`, `foreground`, `relief`, `spacing`-опции, `tabs` и `wrap`.

В следующем примере текстовый виджет слева содержит список URL, по которым можно щелкать мышью. При щелчке определяется нижележащий URL и показывается в текстовом виджете справа.

Листинг 9.91. Файл hyperref.py. Гипертекстовые ссылки средствами Tkinter

```
from Tkinter import *
import urllib
tk = Tk()
txt = Text(tk, width=35)
txt.grid(row=0, column=0, rowspan=2)
show_addr=Text(tk, background="White", width=35, height=1)
show_addr.grid(row=0, column=1)
show_page=Text(tk, background="White", width=35)
show_page.grid(row=1, column=1)

def show_url(event):
  click_point = "@%s,%s" % (event.x, event.y)
  tr = txt.tag_ranges("url")  # список отмеченных областей текста
  url = ""
  # определяем, какой области принадлежит URL
  for i in range(0, len(tr), 2):
    if txt.compare(tr[i], "<=", click_point) and \
       txt.compare(click_point, "<=", tr[i+1]):
      url = txt.get(tr[i], tr[i+1])
  show_addr.delete("1.0", END)
  show_addr.insert("1.0", url) # показываем сам URL
  show_page.delete("1.0", END) # и соответствующий ему документ
  show_page.insert("1.0", urllib.urlopen(url).read())

urls = ["http://www.python.org", "http://www.jython.org"]
for u in urls:
  txt.insert(END, u, "url")  # помещаем URL в текст
  txt.insert(END, "\n")      # конец строки не помечен
# ссылки отмечаем подчеркиванием и синим цветом
txt.tag_config("url", foreground="Blue", underline=1)
# при щелчке мыши на тексте, отмеченном тегом "url" вызывать show_url
txt.tag_bind("url", "<1>", show_url)

tk.mainloop()          # запускаем цикл событий
```

Рисунок (виджет *Canvas*)

Виджет рисунок (`Canvas`*)* является контейнером для различных примитивных графических элементов: геометрических фигур, растровых изображений (от пиктограмм до фото), текста, виджетов. С этими элементами можно проводить определенные манипуляции, их можно перемещать с места на место (в рамках рисунка), раскрашивать в различные цвета, связывать с ними реакцию на определенные события. Некоторые возможности виджета `Canvas` будут рассмотрены в примерах ниже.

В первом примере (листинг 9.92) мы создаем хаотически движущийся кружок, скоростью которого можно управлять с помощью шкалы. Результат работы этой программы можно увидеть на рис. 9.11.

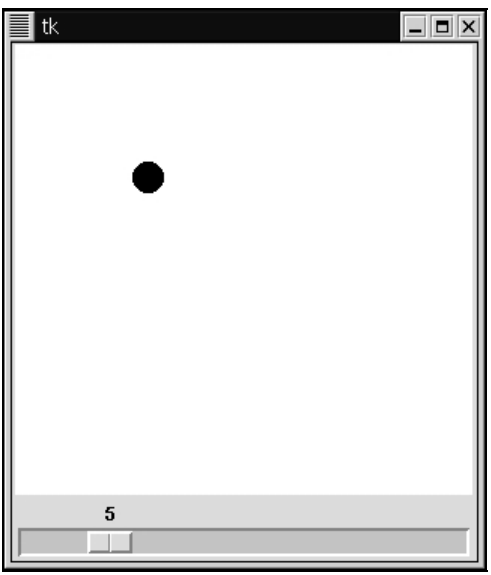


Рис. 9.11. Хаотически движущийся кружок

Листинг 9.92. Файл chaotic.py. Хаотичное движение

```
from Tkinter import *
import whrandom, time

tk = Tk(); f = Frame(tk); f.pack()
# создаем рисунок
c = Canvas(f, bg="White", width="4i", height=300, relief=SUNKEN)
c.pack(expand=1, fill=BOTH)
```

```
# переменной speed
speed = StringVar()
# будет управлять шкала:
slider = Scale(f, from_=0, to=30, orient=HORIZONTAL,
               length="4i", variable=speed)
speed.set("5.0")
slider.pack()

# создаем овал (в данном случае - кружок)
x, y, r = 100, 100, 10
oval = c.create_oval(x-r, y-r, x+r, y+r, fill="Black")

# в цикле приводим кружок в хаотичное движение
for i in range(1000):
  spd = float(speed.get())
  x = x + (whrandom.random() - 0.5)*spd
  y = y + (whrandom.random() - 0.5)*spd
  c.coords(oval, x-r, y-r, x+r, y+r)
  tk.update()  # даем возможность виджетам перерисоваться
  time.sleep(0.1)
```

Наша программа не совсем корректна, т. к. в программе не используется цикл обработки событий, и потому ее сложно включить в состав событийно-ориентированного приложения. В следующем примере (листинг 9.93) приведена программа `chaotic2.py`, которая позволяет пользователю управлять белым квадратиком, в то время как черные кружки хаотически движутся. В программе имеются два класса: `Chaotic`, объекты которого управляют движением кружков, и `Manual`, позволяющий пользователю перетаскивать с помощью мыши белый квадратик.

Листинг 9.93. Файл chaotic2.py. Хаотическое и управляемое движение

```
from Tkinter import *
import whrandom

tk = Tk(); f = Frame(tk); f.pack()

class Chaotic:
  def __init__(self, frame, canvas):
    self.canvas = canvas
```

```
    self.speed = StringVar()
    Scale(frame, from_=0, to=30, orient=HORIZONTAL,
                  length="4i", variable=self.speed).pack()
    self.speed.set("5.0")
    self.x, self.y, self.r = 100, 100, 10
    self.oval = canvas.create_oval(self.x - self.r, self.y - self.r,
      self.x + self.r, self.y + self.r, fill="Black")
    self.canvas.after(10, self.move)

  def move(self):
    """Очередное движение в случайном направлении"""
    spd = float(self.speed.get())
    self.x = self.x + (whrandom.random() - 0.5)*spd
    self.y = self.y + (whrandom.random() - 0.5)*spd
    self.canvas.coords(self.oval, self.x - self.r, self.y - self.r,
      self.x + self.r, self.y + self.r)
    self.canvas.after(10, self.move)

class Manual:
  def __init__(self, frame, canvas):
    self.canvas = canvas
    self.rect = canvas.create_rectangle(50, 50, 70, 70, fill="White")
    # привязываем квадратик к событиям
    # нажатия кнопки мыши (на квадратике)
    canvas.tag_bind(self.rect, "<1>", self.mouseDown)
    # и движения мыши (с нажатой левой кнопкой)
    canvas.tag_bind(self.rect, "<B1-Motion>", self.mouseMove)

  def mouseDown(self, event):
    """Запоминаем начальную точку движения мыши"""
    self.x, self.y = event.x, event.y

  def mouseMove(self, event):
    """Двигаем кружок вместе с мышью. Под CURRENT Tkinter
    понимает объект, вызвавший обрабатываемое событие."""
    self.canvas.move(CURRENT, event.x - self.x, event.y - self.y)
    self.x, self.y = event.x, event.y

c = Canvas(f, bg="White", width="4i", height=300, relief=SUNKEN)
c.pack(expand=1, fill=BOTH)
```

```
chaotic1 = Chaotic(f, c)
chaotic2 = Chaotic(f, c)
manual = Manual(f, c)
tk.mainloop()
```

В приведенном примере для вызова действия через определенное время был использован метод `after()`, первый аргумент которого — время в миллисекундах, а второй — функция или метод, которые нужно вызвать по прошествии этого времени.

Описания методов виджета `Canvas` приведены в табл. 9.146. В этой таблице параметр `bbox` — последовательность координат `(x1, y1, x2, y2)`, задающая ограничивающий прямоугольник соответствующего графического элемента по осям *x* и *y*. Параметр `position` задает координату якоря элемента; `coords` — последовательность координат вершин ломаной или многоугольника; `item` или `id` — целое число, идентификатор графического элемента, или строка с пометкой; `items` — последовательность идентификаторов для групповых операций над элементами (или пометки).

Многие графические элементы имеют опции, приведенные в табл. 9.145.

Таблица 9.145. *Некоторые опции графических элементов виджета* `Canvas`

Опция	Описание
`fill`	Цвет заливки внутренней площади элемента
`stipple`	Растровое изображения шаблона заливки (не поддерживается на Windows)
`outline`	Цвет границы
`outlinestipple`	Растровое изображения шаблона границы (не поддерживается на Windows)
`width`	Ширина границы (по умолчанию в пикселах)
`tags`	Одна или несколько (кортеж) строк с пометками (позволяет группировать элементы с одинаковыми пометками для выполнения некоторых операций)
`anchor`	Расположения якоря элемента относительно самого элемента. Принимает значения сторон света `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW` или `CENTER`

Таблица 9.146. *Методы виджета* `Canvas`

Метод	Описание	Некоторые опции создаваемого объекта
`create_arc(bbox, опции) -> id`	Создает дугу	`style` принимает значения `PIESLICE` (сектор), `CHORD` (хорда) или `ARC` (дуга); `start` — начальный угол (от положительного направления оси x); `extent` — угол дуги (против часовой стрелки)
`create_bitmap (position, опции) -> id`	Создает элемент — битовую маску	`foreground` задает цвет ненулевых пикселов, а `background` — нулевых
`create_image (position, опции) -> id`	Создает элемент — графическое изображение	`image` — объект `PhoteImage`, `BitmapImage` или соответствующие классы из библиотеки PIL *(см. гл. 12)*
`create_line (coords, опции) -> id`	Создает ломаную линию	`arrow` задает направление стрелки (`NONE` (нет стрелки), `FIRST` (у первой точки), `LAST` (у последней точки), `BOTH` (в обоих концах)); `smooth` — параметр сглаживания (по умолчанию `0`). Имеются и другие опции
`create_oval (bbox, опции) -> id`	Создает овал	`fill`, `stipple`, `outline`, `width`, `tags`
`create_polygon(coords, опции) -> id`	Создает многоугольник	`smooth` (то же значение, что и у `create_line()`)
`create_rectangle( bbox, опции) -> id`	Создает прямоугольник	
`create_text (position, опции) -> id`	Создает текстовый элемент	`font` (шрифт), `justify` (выравнивание), `text` (строка с текстом)
`create_window (position, опции) -> id`	Создает окно (в окне можно располагать виджеты). Окно всегда рисуется поверх других элементов рисунка	`window` — встраиваемый виджет; `height`, `width` — высота и ширина

`delete(items)`	Удаляет элементы `items`

Таблица 9.146 *(продолжение)*

Метод	Описание	Некоторые опции создаваемого объекта
`find_closest (x, y) -> item`	Возвращает элемент, ближайший к точке `x, y`	
`itemcget(item, option) -> s`	Возвращает значение опции `option` элемента `item`	
`itemconfig (item, опции), itemconfigure (item, опции)`	Изменяет одну или несколько опций всех заданных элементов	
`coords(item) -> координаты`	Возвращает координаты заданного элемента	
`coords(item, координаты)`	Изменяет координаты заданного элемента	
`bbox(items) -> bbox`	Возвращает (приблизительные) координаты ограничивающего прямоугольника для заданных элементов	
`canvasx(xscr) -> x`	Преобразует оконную координату `xscr` в координату на рисунке	
`canvasy(yscr) -> y`	То же для координаты `yscr`	
`gettags(item) -> tuple`	Возвращает пометки, ассоциированные с данным элементом	
`tag_bind (item, seq, func[, "+"])`	Заменяет (добавляет) привязку элемента `item` к событию `seq`. В случае использования в качестве `item` пометки, на события будут реагировать только элементы, которые имеют пометку, но не вновь создаваемые элементы с той же пометкой *(см. статью "Описания событий" в разд. 9.19.2)*	
`tag_unbind (item, seq)`	Снимает привязку события `seq` к элементу `item`	
`type(item) -> s`	Возвращает тип элемента `item` в виде строки	

`lift(item)`	Пододвигает элемент (или элементы) ближе к переднему плану рисунка	

Таблица 9.146 *(окончание)*

Метод	Описание	Некоторые опции создаваемого объекта
`lower(item)`	Отодвигает элемент (элементы) на задний план	
`move(item, dx, dy)`	Сдвигает элемент `item` на `dx`, `dy` единиц по соответствующим осям	
`scale(item, dx, dy, xscale, yscale)`	Увеличивает `item` в `xscale`, `yscale` раз по соответствующим осям. Перед преобразованием элемент двигается на `-dx`, `-dy`, а после преобразования — двигает обратно на ту же величину	

Помимо описанных выше методов, виджет `Canvas` имеет много других методов, которые позволяют находить элементы в зависимости от их расположения друг к другу (`find_above`, `find_below`), попадания в заданный прямоугольник (`find_enclosed`, `f:find_overlapping`). Виджет `Canvas` имеет также методы для работы с текстовыми элементами, прокруткой и метод для печати рисунка.

Пример в листинге 9.94 показывает применение ломаной для вывода графика функции синус, а на рис. 9.12 представлен результат работы этой программы.

Листинг 9.94. Файл sin.py. Построение графика функции с помощью ломаной линии

```
from Tkinter import *
import math

tk = Tk(); tk.title("График функции"); f = Frame(tk); f.pack()
c = Canvas(f, bg="White", width=300, height=200)
c.pack(expand=1, fill=BOTH)
line = []
for x in range(1, 290, 2):
  line.extend([x+5, 100-int(math.sin(x/25.)*50)])
```

```
c.create_line(line)
c.create_line(0, 100, 290, 100, arrow="last")
c.create_line(5, 10, 5, 190, arrow="first")
c.create_text(90, 60, text="y=sin(x)")
c.create_text(165, 100, text="2p", font="Symbol", anchor=NW)
c.create_text(9, 100, text="0", font="Symbol", anchor=NW)
tk.mainloop()
```

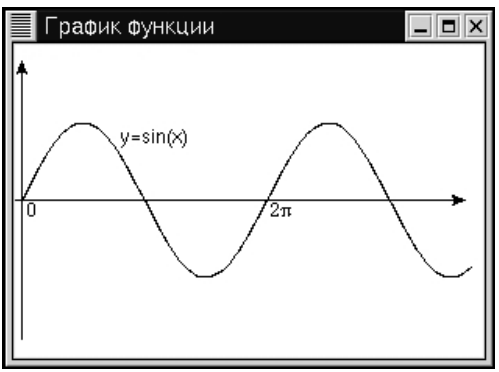


Рис. 9.12. График функции синус, построенный программой sin.py

Менеджеры расположения

Для расположения виджетов внутри другого виджета в `Tkinter` можно пользоваться тремя видами *менеджеров расположения*: `Pack`, `Grid`, `Place`. Для удобства изложения мы назовем *виджетом-хозяином (master widget)* тот виджет, в который встраивается *подчиненный виджет (slave widget)*.

Примечание

Никогда не используйте два разных менеджера расположения для одного и того же виджета, т. к. в этом случае возникает эффект лебедя, рака и щуки. Создавайте отдельные рамки, если вам необходимы услуги разных менеджеров расположения.

Все три имеющихся в `Tkinter` менеджера добавляют в виджет методы, перечисленные в табл. 9.147. Вместо `менеджер` можно подставить соответственно `pack`, `grid` или `place`.

Таблица 9.147. *Методы виджетов для управления расположением*

Метод	Описание
менеджер(опции), менеджер_configure(опции)	Располагает виджет в соответствии с опциями

Таблица 9.147 *(окончание)*

Метод	Описание
менеджер_forget()	Убирает виджет. Позднее его можно снова расположить, например, с помощью другого менеджера
менеджер_info()	Возвращает словарь с действующими опциями

Менеджер расположения *Pack*. Этот менеджер позволяет располагать подчиненные виджеты в рамке виджета-хозяина в соответствии с заданными сторонами, к которым подчиненный виджет должен примыкать.

Таблица 9.148. *Опции менеджера* `Pack`

Опция	Описание
side	Указывает, к какой стороне виджета-хозяина прикладывать располагаемый виджет: TOP (к верхней, по умолчанию), LEFT (к левой), BOTTOM (к нижней) и RIGHT (к правой). В рамках одного и того же виджета-хозяина можно пользоваться различными сторонами для различных подчиненных виджетов
fill	Указывает, что подчиненный виджет должен занять все предоставленное пространство. Принимает значения NONE (опция отключена), X, Y или BOTH (по обеим осям). По умолчанию опция отключена
expand	Указывает, что подчиненный виджет должен занять все возможное дополнительное пространство в виджете-хозяине
in_	Явно указывает виджет-хозяин. Виджет-хозяин должен быть либо родительским виджетом, либо быть прямым или опосредованным подчиненным виджетом родительского виджета

Менеджер расположения *Grid*. Расположение в ячейках сетки, предлагаемое менеджером расположения Grid, является, наверное, наиболее популярным методом компоновки элементов интерфейса. Менеджер расположения Grid располагает виджеты как в отдельных ячейках, так и в прямоугольном блоке ячеек. Список опций приведен в табл. 9.149.

Таблица 9.149. *Опции менеджера* `Grid`

Опция	Описание
columnspan	Количество соседних колонок, которые должен занимать виджет
column	Номер колонки (по умолчанию 0)
ipadx, ipady	Дополнительные зазоры внутри виджета

Таблица 9.149 *(окончание)*

Опция	Описание
padx, pady	Дополнительные зазоры между виджетом и границами ячейки
rowspan	Количество соседних строк, которые должен занимать виджет
row	Номер строки (по умолчанию 0)
sticky	Указывает, каким образом расширить виджет, если ячейка, в которой он находится, больше, чем его естественный размер. Может быть любым сочетанием констант W, E, N, S (указывающих стороны света). В частности, W+E+N+S говорит менеджеру расположения о том, что виджет должен быть расширен во все стороны до полного заполнения ячейки
in_	Явно указывает виджет-хозяин. Виджет-хозяин должен быть либо родительским виджетом, либо быть прямым или опосредованным подчиненным виджетом родительского виджета

Менеджер расположения *Place*. Менеджер Place может располагать виджеты в соответствии с точными абсолютными или относительными координатами. Список опций приведен в табл. 9.150. Все относительные величины задаются относительно виджета-хозяина.

Таблица 9.150. *Опции менеджера* Place

Опция	Описание
anchor	Расположение якоря виджета относительного его самого. Может принимать значения: N, NE, E, SE, S, SW, W, NW или CENTER
bordermode	Опция, принимающая значения INSIDE (внутри) и OUTSIDE (снаружи), для указания того, относительно каких границ виджета-хозяина указаны координаты и размеры размещаемого подчиненного виджета. По умолчанию INSIDE
in_	Явно указывает виджет-хозяин. Виджет-хозяин должен быть либо родительским виджетом, либо быть прямым или опосредованным подчиненным виджетом родительского виджета

relwidth, relheight	Относительная ширина и высота виджета
relx, rely	Координаты якоря подчиненного виджета относительно виджета-хозяина
width, height	Ширина и высота
x, y	Абсолютная позиция виджета в пикселах

В листинге 9.95 приведен пример работы различных менеджеров расположения в трех соседних фреймах. Результат выполнения примера показан на рис. 9.13.

Листинг 9.95. Три менеджера расположения бок о бок

```
from Tkinter import *
tk = Tk()
# создаем три рамки
def getFrame(master):
  f = Frame(master)
  f.config(width=100, height=200, bg="White")
  f.pack(side=LEFT, fill=BOTH)
  return f

f1, f2, f3 = getFrame(tk), getFrame(tk), getFrame(tk)

# в первой рамке используется Pack
# Сначала две кнопки лепятся к левому краю,
b1_1 = Button(f1, text="1.1")
b1_1.pack(side=LEFT, fill=BOTH, expand=1)
b1_2 = Button(f1, text="1.2")
b1_2.pack(side=LEFT, fill=BOTH, expand=1)
# затем еще две к нижнему краю
b1_3 = Button(f1, text="1.3")
b1_3.pack(side=BOTTOM, fill=Y)
b1_4 = Button(f1, text="1.4")
b1_4.pack(side=BOTTOM, fill=BOTH)

# во второй рамке используется Grid.
# две кнопки сверху
b2_1 = Button(f2, text="2.1")
b2_1.grid(row=0, column=0, sticky=NW+SE)
b2_1 = Button(f2, text="2.2")
b2_1.grid(row=0, column=1, sticky=NW+SE)
```

```
# и одна на две колонки -- снизу
b2_1 = Button(f2, text="2.3")
b2_1.grid(row=1, column=0, columnspan=2, sticky=NW+SE)

# в третьей рамке используется Place
# Кнопка 3.1 высотой и шириной в пол-рамки, якорь в левом верхнем
# углу, координаты якоря 1/10 от ширины и высоты рамки
b3_1 = Button(f3, text="3.1")
b3_1.place(relx=0.1, rely=0.1, relwidth=0.5, relheight=0.5, anchor=NW)
# Кнопка такого же размера, строго по центру
b3_2 = Button(f3, text="3.2")
b3_2.place(relx=0.5, rely=0.5, relwidth=0.5, relheight=0.5,
           anchor=CENTER)
# Аналогично 3.1, но якорь по центру
b3_3 = Button(f3, text="3.3")
b3_3.place(relx=0.1, rely=0.1, relwidth=0.5, relheight=0.5,
           anchor=CENTER)

tk.mainloop()
```

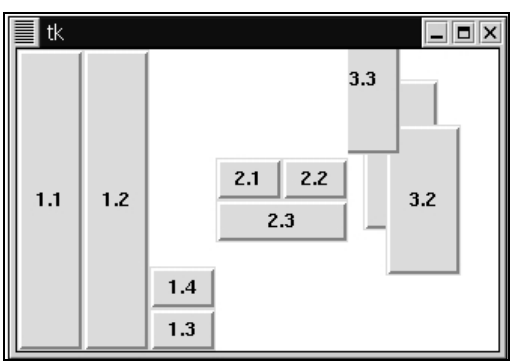


Рис. 9.13. Результат работы различных менеджеров расположения

Переменные

Для работы графического интерфейса недостаточно использования обычных переменных, имеющихся в языке Python, т. к. некоторые переменные являются управляющими элементами, обоюдно связанными с некоторыми характеристиками виджетов. Например, виджет шкала (`Scale`) имеет опцию

variable, которую можно связать с переменной числового типа. После этого установка пользователем значения на шкале приводит к изменению значения в переменной и, наоборот, изменение значения переменной приводит к изменениям на шкале.

В табл. 9.151 перечислены классы переменных, определенных в Tkinter.

Таблица 9.151. *Классы переменных*

Класс	Описание
BooleanVar	Переменная логического типа, принимает два значения: "истина" и "ложь"
DoubleVar	Число с плавающей запятой
IntVar	Целое число
StringVar	Строковая переменная

В табл. 9.152 перечислены методы для работы с экземплярами переменных.

Таблица 9.152. *Методы объектов-переменных модуля* Tkinter

Метод	Описание
get() -> v	Получение значения переменной
set(v)	Установка значения переменной
trace(mode, func) -> funcid	Установка функции слежения за переменной. Параметр mode (режим) — сочетание "r" (чтение переменной), "w" (запись в переменную), "u" (удаление переменной), в зависимости от того, какие операции над переменной необходимо отслеживать. За переменной могут следить несколько функций. У метода trace имеется синоним: trace_variable. Метод возвращает имя Tk-функции, соответствующей func (в дальнейшем это имя можно использовать в trace_vdelete()). Функция func получает три аргумента
trace_vdelete (mode, funcid)	Удаляет функцию слежения с внутренним именем funcid и режимом mode
trace_vinfo() -> lst	Возвращает список кортежей с вида (mode, funcid) для всех добавленных к данной переменной функций слежения

При исполнении программы в листинге 9.96 в окне показывается текущее время.

Листинг 9.96. Файл dclock.py. Электронные часы

```
from Tkinter import *
import time

tk = Tk(); f = Frame(); f.pack()
time_var = StringVar()
time_label = Label(f, textvariable=time_var, font="Courier 60",
                   bg="Black", fg="#00B000")
time_label.pack()

def tick():
  """Обновление табло электронных часов"""
  t = time.localtime(time.time())
  if t[5] % 2:  # эффект мигающего двоеточия
    fmt = "%H:%M"
  else:
    fmt = "%H %M"
  time_var.set(time.strftime(fmt, t))
  time_label.after(500, tick)  # следующий tick через 0.5 с

time_label.after(500, tick)
tk.mainloop()
```

9.19.4. Черепашья графика: модуль *turtle*

Модуль `turtle` содержит возможности для работы с "черепашьей графикой", обязанной своему названию языку Logo. Основной объект — черепашка — приводится в движение простейшими командами, вычерчивая линии (или не вычерчивая, если перо (pen) поднято).

В табл. 9.153 приводится перечень команд для черепашки.

Таблица 9.153. *Команды для черепашки*

Функция	Описание
`turtle.backward(d)`	Назад на `d` пикселов
`turtle.circle(radius[, extent])`	Рисует окружность (или круг). Если угол `extent` задан, то рисует дугу или сектор с заданным углом. Сектор или круг рисуются, если включен режим заливки
`turtle.clear()`	Удаляет все нарисованное
`turtle.color(...)`	Задает цвет в виде строки (с названием или `#`-обозначением) или кортежа из трех цветовых компонентов `(красный, зе-`

	леный, синий). Например, #00FFFF задает желтый цвет
turtle.degrees()	Переключает угловую меру на градусы
turtle.down()	Опускает перо
turtle.fill (flag)	Включает (flag равен 1) или выключает (flag равен 0) режим заливки

Таблица 9.153 *(окончание)*

Функция	Описание
turtle.forward(d)	Вперед на d пикселов
turtle.goto(x, y)	Переход в точку (x, y)
turtle.left(a)	Поворот влево на угол a
turtle.radians()	Переключает угловую меру на радианы
turtle.reset()	Приводит поле и черепашку в начальное состояние
turtle.right(a)	Поворот вправо на угол a
turtle.tracer (flag)	Управляет показом промежуточных положений черепашки (если flag равен 1 — показывать, иначе не показывать)
turtle.up()	Поднимает перо
turtle.width(w)	Устанавливает ширину вычерчиваемых черепашкой линий в значение w
turtle.write (text[, move])	Пишет text. Если флаг move задан и не ноль, черепашка двигается в соответствии с написанным текстом

Приведенных команд достаточно, чтобы рисовать. Следующий пример в листинге 9.97 рисует простейший *фрактал*. Фракталы — объекты, в которых части подобны целому, поэтому для их рисования очень подходит рекурсия *(см. разд. 3.3)*.

Листинг 9.97. Файл fractal.py. Рисование фрактала

```
import turtle

def draw_fractal(L, level):
  """Функция для рекурсивного рисования фрактала"""
  if level == 0:
    turtle.forward(L)
  else:
    L1 = L/4.0
    draw_fractal(L1, level-1)
```

```
    turtle.left(30)
    draw_fractal(L1, level-1)
    turtle.right(60)
    draw_fractal(L1, level-1)
    turtle.left(30)
    draw_fractal(L1, level-1)
turtle.reset()
turtle.up()
turtle.backward(100)
turtle.down()
turtle.tracer(1)
draw_fractal(256, 5)
turtle._root.mainloop()
```

Результат работы приведен на рис. 9.14.

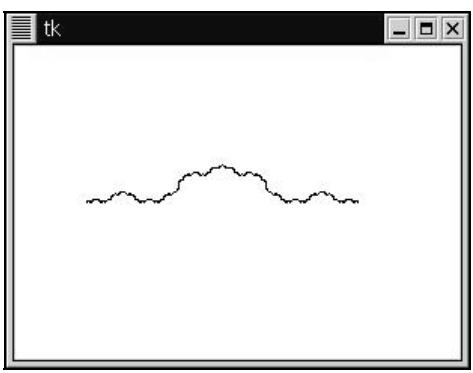


Рис. 9.14. Нарисованный фрактал

9.20. Модули для MS Windows

В этом разделе рассматриваются модули, специфичные для системы Microsoft Windows.

9.20.1. Модуль *msvcrt*

Модуль `msvcrt` (от Microsoft Visual C Runtime Library) дает возможность использовать в программах на Python некоторые функции на платформе Windows. В табл. 9.154 приведены функции, определяемые в этом модуле.

Таблица 9.154. *Функции модуля* `msvcrt`

Функция	Описание
`get_osfhandle (fd) -> handle`	Получает файловый манипулятор по файловому дескриптору `fd`

Таблица 9.154 *(окончание)*

Функция	Описание
`getch() -> char`	Возвращает символ, соответствующий нажатой клавише (ждет нажатия клавиши, если буфер пуст). Если нажата специальная или функциональная клавиша, возвращает `"\000"` или `"\xe0"`, а код клавиши можно получить следующим вызовом `getch()`
`getche() -> char`	То же, но печатает символ на консоли, если он может быть напечатан
`heapmin()`	Отдает неиспользуемую память операционной системе (Windows NT)
`kbhit() -> 0, 1`	Возвращает истину, если в буфере есть код нажатой клавиши
`locking(fd, mode, nbytes)`	Запирает файл с файловым дескриптором `fd`. Возбуждает исключение `IOError` в случае неудачи
`open_osfhandle (handle, flags) -> fd`	Получает файловый дескриптор по файловому манипулятору `handle`. Флаги `flags` могут быть побитовым ИЛИ `os.O_APPEND`, `os.O_RDONLY`, `os.O_TEXT`
`putch(char)`	Выводит `char` на консоль без буферизации
`setmode(fd, flags)`	Устанавливает режим обработки концов строк для файлового дескриптора `fd`. Флаг `os.O_TEXT` включает текстовый режим, `os.O_BINARY` — бинарный
`ungetch(char)`	Заталкивает символ `char` обратно в очередь. В дальнейшем он может быть прочитан функциями `getch()` и `getche()`

9.20.2. Модуль *_winreg*

Модуль `_winreg` появился в Python 2.0 и предоставляет низкоуровневый доступ к системному реестру (registry) ОС Windows. В качестве манипулятора в `_winreg` используется не целочисленное значение, а специальный объект, который обозначается в описаниях ниже как `hkey` и может использоваться на месте аргумента по имени `key`. Под `key` понимается ранее открытый ключ манипулятора или предопределенная в модуле `_winreg.HKEY_`-константа, где это не оговорено особо. Аргумент `sub_key` —

текстовая строка, обозначающая подключ для данного ключа. Остальные обозначения объяснены вместе с описаниями функций. В случае неудачи функции модуль возбуждает исключение `EnvironmentError`.

- `_winreg.CloseKey(key)`

  Закрывает открытый ранее ключ `key` реестра.

- `_winreg.ConnectRegistry(computer, key) -> hkey`

  Устанавливает соединение с предопределенным реестровым манипулятором `key` на удаленном компьютере и возвращает объект-манипулятор `hkey`. Аргумент `computer` — имя удаленного компьютера в форме `\\computername` или `None` для локального компьютера. Аргумент `key` может быть `_winreg.HKEY_`-константой.

- `_winreg.CreateKey(key, sub_key) -> hkey`

  Создает или открывает указанный реестровый ключ.

- `_winreg.DeleteKey(key, sub_key)`

  Удаляет указанный ключ. Значение `sub_key` не должно быть `None`, а удаляемый ключ не должен иметь подключей.

- `_winreg.DeleteValue(key, value)`

  Удаляет значение `value` (строка, идентифицирующая значение) из реестрового ключа.

- `_winreg.EnumKey(key, index) -> sub_key`

  Перечисляет подключи ключа `key`, возвращая строку, соответствующую подключу с номером `index`. Если подключ с данным индексом отсутствует, возбуждается `EnvironmentError`.

- `_winreg.EnumValue(key, index) -> (value_name, value_data, data_type)`

  Перечисляет значения подключей открытого реестрового ключа, возвращая для ключа с номером `index` кортеж из трех элементов. В этом кортеже `value_name` — строка с именем, `value_data` — объект со значением, соответствующим имени, `data_type` — целое число, обозначающее тип данных значения.

- `_winreg.FlushKey(key)`

  Принудительно записывает на диск все произведенные изменения ключа `key` (не возвращается, пока изменения не будут записаны), что обычно не требуется.

- `_winreg.RegLoadKey(key, sub_key, file_name)`

  Создает подключ для данного ключа и сохраняет реестровую информацию из файла с данным в `file_name` именем. Файл должен был быть создан с помощью функции `_winreg.SaveKey()`. Если файл хранится в

файловой системе FAT, он не должен иметь расширения. Путь к файлу определяется относительно имени удаленного компьютера, если key был получен вызовом ConnectRegistry().

- _winreg.OpenKey(key, sub_key[[, res], sam]) -> hkey

  Открывает заданный ключ и возвращает объект-манипулятор. Аргумент res должен быть равен нулю. Аргумент sam — целое число, задающее маску для желаемого режима безопасности доступа к ключу, по умолчанию: KEY_READ. Синоним OpenKeyEx().

- QueryInfoKey(key) -> (num_subkeys, num_values, last_modified)

  Возвращает информацию о ключе в виде кортежа. Здесь num_subkeys — целое число, количество подключей, num_values — целое число, количество значений ключа, last_modified — время последнего изменения ключа (если вообще передается).

- QueryValueEx(key, value_name) -> (value, type_id)

  Возвращает тип и данные указанного имени value_name значения ключа key. Результат — кортеж, состоящий из значения элемента реестра value и целого числа type_id, указывающего тип.

- SaveKey(key, file_name)

  Сохраняет ключ key и все его подключи в файле file_name. Файл с именем file_name не должен существовать до выполнения функции, а при использовании файловой системы FAT он не должен иметь расширение.

- SetValue(key, sub_key, type, value)

  Сохраняет значение value в ключе key, который был ранее открыт с признаком KEY_SET_VALUE. Параметр type может быть только REG_SZ (см. табл. 9.155). Если подключ sub_key не существует, функция его создаст.

- SetValueEx (key, value_name, reserved, type, value)

  То же, но позволяет указывать другие типы данных, перечисленные в табл. 9.155.

Таблица 9.155. *Типы данных значений реестровых ключей*

Целочисленная константа для обозначения типа	Описание
_winreg.REG_BINARY	Произвольные бинарные данные
_winreg.REG_DWORD_BIG_ENDIAN	То же, сначала старшие байты
_winreg.REG_DWORD_LITTLE_ENDIAN	То же, сначала младшие байты
_winreg.REG_DWORD	32-разрядное число

_winreg.REG_EXPAND_SZ	Строка, оканчивающаяся нулем с неподставленными переменными окружения (например, "%PATH%")
_winreg.REG_LINK	Символическая ссылка в Unicode
_winreg.REG_MULTI_SZ	Последовательность строк
_winreg.REG_NONE	Неопределенный тип

Таблица 9.155 *(окончание)*

Целочисленная константа для обозначения типа	Описание
_winreg.REG_RESOURCE_LIST	Список ресурсов драйвера устройства
_winreg.REG_SZ	Строка, оканчивающаяся нулем

В следующем примере из системного реестра извлекается информация о временной зоне:

Листинг 9.98. Файл show_tz.py. Работа с системным реестром

```
from _winreg import *
local = ConnectRegistry(None, HKEY_LOCAL_MACHINE)
subkey = r"SYSTEM\CurrentControlSet\Control\TimeZoneInformation"
e = OpenKey(local, subkey, 0, KEY_READ)
try:
  i = 0
  while 1:
    name, val, type_id = EnumValue(e, i)
    if type_id == REG_SZ:
      print "%4i %26s %2i: %s" % (i, name, type_id, val)
    else:
      print "%4i %26s %2i: %r" % (i, name, type_id, val)
    i = i + 1
except EnvironmentError:
  print "Total: ", i
CloseKey(local)
```

Результат выполнения:

```
   0 DisableAutoDaylightTimeSet  4: 1
   1             ActiveTimeBias  3: '\x10\xff\xff\xff'
```

```
   2              StandardName  1: Московское время (зима)
   3              DaylightName  1: Московское время (лето)
   4                      Bias  3: 'L\xff\xff\xff'
   5              StandardBias  3: '\x00\x00\x00\x00'
   6              DaylightBias  3: '\xc4\xff\xff\xff'
   7             StandardStart  3:
'\x00\x00\n\x00\x00\x00\x05\x00\x03\x00\x00\x00\x00\x00\x00\x00'
   8             DaylightStart  3:
'\x00\x00\x03\x00\x00\x00\x05\x00\x02\x00\x00\x00\x00\x00\x00\x00'
   9              DaylightFlag  3: '\x02\x00\x00\x00'
Total:  10
```

Примечание

Значение ключа может быть Unicode-строкой. В этом случае может случиться, что напечатать его с помощью `print` не удастся, и будет необходимо применить метод `encode()`, задав ему целевую кодировку (например, `"cp866"` или `"cp1251"`).

9.20.3. Модуль *winsound*

С помощью функций модуля `winsound` можно заставить программы на Python звучать. Модуль содержит две функции.

- `winsound.Beep(hz, millisecs)`

  Издает на встроенном динамике звук частотой `hz` от 37 до 32 767 Гц и продолжительностью `millisecs` миллисекунд. Если подача звукового сигнала невозможна, возбуждается исключение `RuntimeError`.

- `winsound.PlaySound(sound, flags)`

  Воспроизводит звук `sound`, заданный в виде файла, строки с бинарными данными или названия системного звука. Флаги `flags` определяют многочисленные аспекты воспроизведения звука.

В табл. 9.156 приведены возможные флаги для функции `winsound.PlaySound()`.

***Таблица 9.156.** Флаги, применяемые в `winsound.PlaySound()`*

Константа	Значение
`SND_FILENAME`	Параметр `sound` является именем файла
`SND_ALIAS`	Параметр `sound` является названием системного звука
`SND_LOOP`	Зациклить воспроизводимый фрагмент

SND_MEMORY	Параметр sound является строкой, содержащей данные в формате WAV
SND_PURGE	Останавливает все проигрываемые экземпляры указанного звука
SND_ASYNC	Начинает воспроизведение и возвращается. Не может использоваться совместно с SND_MEMORY
SND_NODEFAULT	В случае отсутствия указанного системного звука не проигрывать звук по умолчанию

Таблица 9.156 *(окончание)*

Константа	Значение
SND_NOSTOP	Не прерывать воспроизводящиеся звуки
SND_NOWAIT	Возвратиться, если звуковой драйвер занят

В следующем примере показано использование модуля winsound.

```
from winsound import *
Beep(5000, 1000)
wavdate = open("c:/windows/media/start.wav", "rb").read()
PlaySound(wavdate, SND_MEMORY)
Beep(5000, 1000)
```

9.21. Интернационализация и локализация

Исторически сложилось, что современные компьютеры и программное обеспечение хорошо адаптированы к потребностям англоговорящих пользователей и программистов. Конечно, СЛЖ, ВЧТ, ВЫХ (которые можно найти в старых советских книжках) вызывают улыбку, и вместо них специалистам куда привычнее писать ADD, SUB и EXIT, но есть вещи, отсутствие которых делает разработку программ очень неудобным.

Модули, описываемые в этом разделе, поддерживают *интернационализацию* (то есть поддержку многих языков) и *локализацию* (адаптацию интернационализированной программы к конкретному местному языку и культурным традициям).

9.21.1. Модуль *locale*

Было бы неплохо, чтобы в программе можно было использовать те же функции для английских и русских строк. К счастью, такой механизм давно существует и называется *культурная среда (locale)*. Python по большей части использует механизм locale языка C. В правильно настроенной на русскую среду и кодировку koi8-r программе мы наблюдаем следующее:

```
>>> import re, string, locale
>>> print locale.setlocale (locale.LC_ALL, "ru_RU.KOI8-R")
ru_RU.KOI8-R
>>> print string.letters
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzёЁюабцдефгхийклмноп
ярстужвьызшэщчъЮАБЦДЕФГХИЙКЛМНОПЯРСТУЖВЬЫЗШЭЩЧЪ
>>> print string.upper("малые")
МАЛЫЕ
>>> print re.match("малые", "МАЛЫЕ", re.IGNORECASE)
None
>>> print re.match("малые", "МАЛЫЕ", re.IGNORECASE|re.LOCALE)
<SRE_Match object at 0x80f8410>
>>> print re.match("\w+", "слово", re.I)
None
>>> print re.match("\w+", "слово", re.I|re.L)
<SRE_Match object at 0x811ed08>
>>> a = ["юг", "восток", "север", "запад"]
>>> a.sort()
>>> print string.join(a)
юг север восток запад
>>> a.sort(locale.strcoll)
>>> print string.join(a)
восток запад север юг
```

Пример использует модуль `locale` и кодировку koi8-r (именно на эту кодировку была настроена система у автора). В этой кодировке буквы кириллицы расположены не по алфавиту, что вызывает сложности при *сортировке строк* в лексикографическом порядке. Однако механизм locale позволяет решить эту проблему, как это и показано в примере. Видно, что после установки locale с помощью функции `setlocale()`, Python стал считать буквами (letters) не только латинские буквы, но и 33 буквы русского алфавита. Соот-

ветственно, функция `string.upper()` *(см. разд. 9.3.1)* смогла поменять регистр букв, а функция `re.match()` смогла сравнить две строки с игнорированием регистра букв. Точно так же русские буквы вошли в определение "`\w`" только после указания флага. Такие предосторожности оправданы. Например, многие программы разбирают дату и время только в стандартном, английском написании. Задание locale может нарушить функции печати даты и времени (например, вместо стандартных "Tue Feb 6 23:55:06 MSK 2001" может получиться "Втр Фев 6 23:54:58 MSK 2001").

Разберем теперь некоторые функции модуля.

Функция `locale.getdefaultlocale()` определяет культурную среду, в которой запущена программа, и возвращает соответствующий кортеж (или список). Функция `locale.getlocale()` определяет locale для конкретной категории (`LC_MONETARY`, `LC_MESSAGES` и т. п.), по умолчанию `LC_CTYPE`.

```
locale.getdefaultlocale() -> [language_code, encoding]
locale.getlocale(category) -> [language_code, encoding]
```

где `language_code` — код языка; `encoding` — кодировка; `category` — категория.

```
import locale
print locale.getdefaultlocale ()
print locale.getlocale (LC_CTYPE)
```

Результат выполнения зависит от стандартных настроек среды:

```
['ru_RU', 'ISO8859-5']
['ru_RU', 'ISO8859-5']
```

или:

```
(None, None)
(None, None)
```

Функция `locale.setlocale()` устанавливает культурную среду для данной категории. Список доступных категорий приведен в табл. 9.157.

```
locale.setlocale(category,locale) -> string
```

где `category` — категория; `locale` — в виде строки, списка `[language_code, encoding]` или `None`; `string` — строка с установленной культурной средой.

Таблица 9.157. *Категории культурной среды*

Категория	Что затрагивает
`LC_ALL`	Все аспекты
`LC_MESSAGES`	Текстовые сообщения
`LC_TIME`	Представление даты и времени

LC_COLLATE	Сравнение строк
LC_MONETARY	Представление денежной единицы
LC_CTYPE	Манипуляции с символами и многобайтными символами
LC_NUMERIC	Представление чисел

Установим кодировку символов koi8-r:

```
import time, locale
locale.setlocale(locale.LC_ALL, None)
print time.strftime("%d %B", time.localtime (time.time()))
locale.setlocale(locale.LC_ALL, "ru_RU.KOI8-R")
print time.strftime("%d %B", time.localtime (time.time()))
```

Результат выполнения:

```
11 February
11 Февраля
```

Функция `locale.str()` аналогична встроенной функции `str()`, но учитывает культурную среду в категории `LC_NUMERIC`.

```
str(number) -> string
```

где `number` — форматируемое число; `string` — строка с представлением числа в данном locale.

```
>>> from locale import *
>>> print str(12.300)
12.3
>>> setlocale (LC_NUMERIC, "ru_RU.KOI8-R")
'ru_RU.KOI8-R'
>>> print str(12.300)
12,3
```

Функция `locale.strcoll()` сравнивает две строки, сообразуясь с установленным `LC_COLLATE`.

```
strcoll(s1, s2) -> int
```

где `s1`, `s2` — сравниваемые строки; `int` — результат сравнения. Отрицательное число, если `s1 < s2`, `0`, если `s1 == s2`, и положительное число, если `s1 > s2`.

```
>>> from locale import strcoll, setlocale, LC_COLLATE
>>> print strcoll("a", "ю"), strcoll("ю", "a")
1 -1
>>> print cmp("a", "ю"), cmp("ю", "a")
```

```
1 -1
>>> print setlocale (LC_COLLATE, "ru_RU.KOI8-R")
ru_RU.KOI8-R
>>> print strcoll("а", "ю"), strcoll("ю", "а")
-1 1
>>> print cmp("а", "ю"), cmp("ю", "а")
1 -1
```

До переключения `LC_COLLATE` на кодировку koi8-r строки с русскими буквами сравнивались неправильно. После переключения на koi8-r — сравнение с помощью `locale.strcoll()` стало корректным (тогда как `cmp()` сравнивает строки по-прежнему неверно).

Функция `locale.format()` позволяет форматировать число для вывода аналогично операции `"%"`, но принимая во внимание установки `LC_NUMERIC`.

❐ `format(fmt, number[, grouping=0]) -> string`

где `fmt` — строка форматирования, аналогична левому аргументу операции `"%"`; `number` — форматируемое число; `grouping` — если принимает значение "истина", выделять группы разрядов.

В отличие от англоязычных стран, в России принято писать в числах вместо плавающей точки запятую (да и называть их числами с плавающей запятой). Следующий пример иллюстрирует различия культурных сред в написании чисел.

```
>>> from locale import format, LC_NUMERIC, setlocale
>>> print format("%f", -12300.8)
-12300.800000
>>> print format("%f", -12300.8, 1)
-12300.800000
>>> print setlocale (LC_NUMERIC , "en_US")
en_US
>>> print format("%f", -12300.8)
-12300.800000
>>> print format("%f", -12300.8, 1)
-12,300.800000
>>> print setlocale (LC_NUMERIC , "ru_RU")
ru_RU
>>> print format("%f", -12300.8)
-12300,800000
>>> print format("%f", -12300.8, 1)
-12.300,800000
```

Функции `locale.atoi()` и `locale.atof()` аналогичны `string.atoi()` и `string.atof()`, за исключением того, что они используют `LC_NUMERIC` для конвертирования строки с записью числа в число.

```
locale.atoi(string) -> number

locale.atof(string) -> number
```

где `string` — строка с записью числа; `number` — прочитанное число.

Функция `atoi()` читает из строки целое число, а `atof()` — число с плавающей точкой. В случае ошибки возбуждается исключение `ValueError`.

```
>>> from locale import *
>>> setlocale (LC_NUMERIC, "ru_RU.KOI8-R")
'ru_RU.KOI8-R'
>>> print atoi("12.000")
12000
>>> print atof("12,000")
12.0
```

Функция `locale.strxfrm()` готовит строку к сравнению с помощью встроенной функции `cmp` согласно установленному `LC_COLLATE`.

```
strxfrm(string) -> string1
```

где `string` — исходная строка; `string1` — строка, которая годится для корректного сравнения с себе подобными.

```
>>> from locale import LC_COLLATE, setlocale, strxfrm
>>> a, b = "юла", "волчок"
>>> print a < b
1
>>> setlocale(LC_COLLATE, "ru_RU.KOI8-R")
'ru_RU.KOI8-R'
>>> print a < b
1
>>> print strxfrm(a) < strxfrm(b)
0
```

Как уже говорилось, буквы в кодировке koi8-r идут не в алфавитном порядке, поэтому правильно сортировать строки в этой кодировке поможет только установка соответствующей культурной среды (locale).

9.21.2. Модуль *gettext*

Модуль `gettext` поддерживает механизмы поддержки многих языков и часть локализации, относящуюся к адаптации программы к местному языку. Для этого программам на Python предоставляется интерфейс к каталогам с сообщениями программы GNU gettext. Модуль использует два подхода к работе с GNU gettext: функции API и объектно-ориентированный доступ.

Легче всего познакомиться с `gettext` на примере. Предположим, что нам необходимо сделать интернациональной следующую программу (листинг 9.99):

Листинг 9.99. Файл iprog-pre.py. Программа для интернационализации

```
import sys
print "Hello, world!"                                  # \/
params = ["help", "version"]                          # \/
print "My parameters:"                                 # \/
for p in sys.argv[1:]:
  if p[:2] == "--" and p[2:] in params:
    print "Parameter --%s gives %s;" % (p, p)          # \/
  else:
    print "Parameter %s;" % p                          # \/
```

В этой программе строки, подлежащие переводу, помечены галочками. В модуле `gettext` строки переводятся с помощью функции `gettext.gettext()`, поэтому все строковые литералы, подлежащие переводу, должны быть охвачены этой функцией. Для того чтобы в дальнейшем автоматически создать файл с шаблоном для переводчика, необходимо, чтобы функция `gettext.gettext()` называлась `_()`. Кроме того, некоторые строковые литералы не должны переводиться на месте их появления в программе, а несколько позже и в другом месте. Такие литералы, чтобы они попали в файл шаблона, нужно охватить другой функцией, назовем ее `n_`. В нашем примере литералы `"version"` и `"help"` не должны переводиться немедленно, т. к. они несут двойную нагрузку: и как слово, и как название опции, которая в любом случае должна остаться на английском. С учетом этого, в листинге 9.100 представлена программа, правильно размеченная для дальнейшей автоматической обработки:

Листинг 9.100. Файл iprog.py. Программа для интернационализации

```
import sys


# Добавляем поддержку gettext:
```

```
import gettext
gettext.bindtextdomain('iprog', 'locale')
gettext.textdomain('iprog')
_ = gettext.gettext
n_ = lambda x: x

print _("Hello, world!")
params = [n_("help"), n_("version")]
print _("My parameters:")
for p in sys.argv[1:]:
  if p[:2] == "--" and p[2:] in params:
    print _("Parameter --%s gives %s;") % (p[:2], _(p[:2]))
  else:
    print _("Parameter %s;") % p
```

Теперь мы можем подготовить pot-файл (файл с шаблоном для переводчика). В комплекте GNU gettext есть программа `xgettext`, которая позволяет извлекать строки, подлежащие переводу, из программ, написанных на C. К сожалению, эта программа не может быть применена к программам на языке Python. Однако в каталоге `Tools/i18n` дистрибутива с исходными кодами интерпретатора Python мы можем найти программу `pygettext.py`, с помощью которой и получим файл-шаблон:

```
pygettext.py -k n_ -o iprog.pot iprog.py
```

С помощью этой программы мы обрабатываем файл iprog.py и записываем pot-файл в `iprog.pot`. Мы также указали с помощью ключа `-k`, что функция `n_()` тоже должна учитываться при извлечении строковых литералов.

Дальше мы должны передать pot-файл переводчикам, чтобы те перевели английские строковые литералы на нужные нам языки. Вот как будет выглядеть часть pot-файла, переведенная на русский (принято давать переведенному файлу расширение po):

Листинг 9.101. Файл iprog.po. Фрагмент файла с переводами строковых литералов

```
#: iprog.py:16
msgid "Parameter %s;"
msgstr "Параметр %s;"

#: iprog.py:10
msgid "help"
msgstr "помощь"
```

```
#: iprog.py:13
msgid "My parameters:"
msgstr "Мои параметры:"
```

Теперь нужно превратить текстовый po-файл в бинарный mo-файл с помощью утилиты `msgfmt` от GNU или `msgfmt.py`, находящейся тоже в `Tools/i18n`:

```
msgfmt.py -o iprog.mo iprog.po
```

Теперь нам нужно положить mo-файл в каталог `locale/ru/LC_MESSAGES` (в каталоге, где лежит программа `iprog.py`) и попробовать запустить программу. Вот как это выглядит в UNIX:

```
$ ./iprog.py
Hello, world!
My parameters:
$ export LANG=ru
$ ./iprog.py --help --version -a
Здравствуй, мир!
Мои параметры:
Параметр --help дает помощь;
Параметр --version дает версию;
Параметр -a;
```

Таким образом, если мы подготовим mo-файлы для всех нужных нам языков, мы сможем вызывать программу с использованием нужного нам языка, задав переменную окружения `LANG`, `LC_MESSAGES` или `LANGUAGE`. Если перевод на требуемый нам язык не найден, сообщение будет выведено на английском.

Итак, для интернационализации программы необходимо сделать:

- охватить строковые литералы, подлежащие переводу, функцией `_()` или `n_()`, а также применить функцию `_()` там, где перевод над исходными литералами не производился;
- использовать программу `pygettext.py` для извлечения строковых литералов, подлежащих переводу, в pot-файл;
- перевести pot-файл на один или несколько языков, получив соответствующие po-файлы;
- использовать программы `msgfmt` или `msgfmt.py` для получения mo-файлов;
- поместить mo-файлы в соответствующие каталоги сообщений.

Таким образом, перевод сообщений программы (одна из главных составляющих локализации) хоть и требует некоторой дополнительной возни, является не таким сложным делом. И, естественно, конечные пользователи вашей программы будут очень благодарны за сообщения на своем родном языке!

Ниже дано описание функций модуля `gettext`.

- `gettext.bindtextdomain(область[, localedir])`

  Связывает `область` с каталогом культурной среды `localedir`. Если каталог опущен или `None`, возвращает текущий каталог культурной среды. Важно, чтобы на пути `язык/LC_MESSAGES/область.mo` находились файлы с переводами, где `язык` задан переменными окружения `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` или `LANG`, а `область` может, например, соответствовать имени программы. В некоторых системах, например Linux, путь к каталогу культурной среды стандартизирован — это `/usr/share/locale` или `/usr/local/share/locale`.

- `gettext.textdomain([область])`

  Устанавливает или получает текущую область сообщений.

- `gettext.gettext(message) -> message1`

  Возвращает локализованное сообщение `message`. Эта функция обычно получает имя `_` в локальном пространстве имен.

- `gettext.dgettext(область, message) -> message1`

  Возвращает локализованное с помощью `области` сообщение `message`.

Более удобным подходом оказывается использование классов, определенных в модуле `gettext`. Этот модуль определяет классы `NullTranslations` и `GNUTranslations` и функции для работы с ними.

- `gettext.find(область[, localedir[, lang_lst]]) -> filename`

  Возвращает первый попавшийся файл с переводами `области` в каталоге `localedir` (по умолчанию: `sys.prefix+"/share/locale"`) для списка языков `lang_lst` (если список языков не задан, язык берется из переменных окружения).

- `gettext.translation(область[, localedir[, lang_lst[, class_]]]) -> tobj`

  Возвращает экземпляр класса `class_` (по умолчанию: `GNUTranslations`), который использует те же параметры, что и `gettext.find()`.

- `gettext.install(область[, localedir[, unicode]])`

Устанавливает функцию `_()` во встроенное пространство имен Python. Параметры те же, что и у предыдущих функций. Ненулевой флаг `unicode` говорит о том, что функция `_()` будет возвращать Unicode-строки.

В следующем примере мы используем все тот же mo-файл:

```
>>> import gettext
>>> os.environ["LANG"] = "ru"
>>> print gettext.find("iprog", "locale", ["ru"])
locale/ru/LC_MESSAGES/iprog.mo
>>> tr = gettext.translation("iprog", "locale")
>>> print tr.gettext("help")
помощь
>>> gettext.install("iprog", "locale", unicode=1)
>>> print _("help").encode("koi8-r")
помощь
```

Замечание

Функцию `_()` в интерактивном режиме можно использовать после `gettext.install()` только один раз, т. к. в `_` помещается предыдущий результат.

Методы объектов `Translation`-классов: метод `gettext()` для получения перевода, `ugettext()` для получения перевода в Unicode, `info()` дает информацию из заголовка po-файла, `charset()` возвращает используемую в mo-файле кодировку.

Благодаря использованию объектно-ориентированного API модуля `gettext` приложение может переключать используемые языки:

```
import gettext
lang_en = gettext.translation("mydomain", languages=['en'])
lang_ru = gettext.translation("mydomain", languages=['ru'])
lang_ru.install()
print _("word")  # будет переведено на русский
lang_en.install()
print _("word")  # английский вариант
```

Примечание

Разумеется, для правильной работы приведенного примера необходим соответствующий mo-файл.

9.22. Используем Distutils

Утилиты Distutils служат для облегчения работы по распространению модулей для Python. С одной стороны, они позволяют разработчику модуля, набора модулей или пакета *(см. разд. 8.8)* создавать установщик или архив с файлами для установки, а пользователю — легко и удобно устанавливать приготовленные таким образом пакеты у себя в системе. Ниже мы рассмотрим оба этих аспекта утилит Distutils.

В дальнейшем файлы, необходимые для распространения и установки, мы будем называть *дистрибутивами*. Дистрибутивы бывают двух видов: *дистрибутивы с исходным кодом* и *бинарные дистрибутивы*. Разница между ними появляется только тогда, когда распространяемые модули написаны не чисто с использованием Python и, значит, требуют компиляции.

Бинарные дистрибутивы привязаны к определенной платформе (Macintosh, конкретной разновидности UNIX, Windows), о чем нужно помнить, выбирая дистрибутив.

Дистрибутив с исходным кодом является основой для построения бинарных дистрибутивов для нужной платформы. Его используют пользователи, которые по той или иной причине не могут или не хотят использовать готовый бинарный дистрибутив. Этими причинами может быть отсутствие бинарного дистрибутива для целевой платформы, намерение лучше оптимизировать бинарный код или просто желание посмотреть исходный код с точки зрения его безопасности (что всегда рекомендуется делать при получении дистрибутива из ненадежного источника).

Замечание

Описание Distutils отличается от описаний других модулей ориентаций на использование в типичных ситуациях. Разработчикам дистрибутивов, возможно, потребуется ознакомиться с оригинальной документацией, чтобы использовать Distutils полностью.

9.22.1. Установка модулей

В случае, когда автор или распространитель модуля изготовили специальный установочный файл для вашей компьютерной платформы и версии Python, вам достаточно просто запустить установщик (например, exe-файл для Windows или sea-файл для Macintosh) или применить стандартную процедуру установки (например, `rpm -i имя-пакета` в RedHat, SuSe или Mandrake Linux). Ниже будет рассмотрен другой случай, когда интересующий вас модуль поставляется в виде дистрибутива с исходными кодами, помещенного в zip- или tar.gz-архив.

Многие новые программные модули, написанные на Python, поставляются с использованием Distutils, что делает их установку достаточно простым делом. В этом разделе мы рассмотрим, как можно использовать Distutils для установки Python-программ в Windows и UNIX. Установка на платформе Macintosh очень похожа на установку в Windows, и мы ее рассматривать не будем.

В среде Windows

Предположим, что у нас есть дистрибутив с исходными кодами нужных модулей: `Combinatorial-1.0.zip`. Для распаковки архива необходима программа типа WinZip, UltimateZip или любая другая, умеющая работать с zip-архивами. Предположим, что содержимое архива развернуто в каталоге `C:\TEMP`. Также подразумевается, что установка самого Python выполнена в соответствии с рекомендациями, которые давались в *разд. 7.5*.

Установка модуля в стандартный каталог для сторонних модулей производится по команде (выполняется из приглашения DOS):

```
C:\TEMP\Combinatorial> python setup.py install
```

Другой вариант состоит в том, что сначала с помощью команды

```
C:\TEMP\Combinatorial-1.0> python setup.py bdist_wininst
```

готовится установщик (exe-файл), а собственно установка производится вызовом этого файла на исполнение.

Несколько сложнее обстоит дело, если в дистрибутиве присутствуют файлы на языке C, требующие компиляции, т. к. в этом случае требуется компилятор языка C. Список подходящих компиляторов дан в табл. 9.160. На прилагаемом к книге компакт-диске есть компилятор mingw32. Конечно, если вы не хотите иметь дело с компиляторами, придется искать бинарный дистрибутив нужных вам модулей-расширений.

В среде UNIX

Python и Distutils наиболее естественны для среды UNIX и Linux и программных средств GNU. Предположим, что у нас есть дистрибутив с исходными кодами нужных модулей: `Combinatorial-1.0.tar.gz`, `Combinatorial-1.0.tar.bz2` или `Combinatorial-1.0.zip`. Для распаковки архивов нужны программы `tar` (для первых двух), `gzip`, `bzip2` или `unzip` для третьего архива. Поместите файл с архивом в некоторый каталог, где можно его распаковать, и примените одну из команд, соответствующую вашему типу архива:

```
$ tar xzf Combinatorial-1.0.tar.gz
$ bzip2 -c -d Combinatorial-1.0.tar.bz2 | tar xf -
$ unzip Combinatorial-1.0.zip
```

В результате должен образоваться каталог `Combinatorial-1.0` со следующими файлами:

```
Combinatorial
|-- Combinatorial.py
|-- setup.cfg
`-- setup.py
```

Установка модуля в стандартный каталог для сторонних модулей производится по команде ("`#`" говорит о том, что установку производит пользователь `root`):

```
# python setup.py install
```

Можно, конечно, произвести установку и в другой каталог. UNIX предлагает для этого две схемы: установка в домашний каталог и установка в произвольный заданный каталог. В любом случае Python может сказать нам, что модули установлены в каталог, который отсутствует в списке путей поиска модулей `sys.path` и потому для использования этих модулей может потребоваться изменение списка путей.

Следующие два примера иллюстрируют оба подхода:

```
$ python setup.py install --home=~
# python setup.py install --prefix=/usr/local
```

В первом случае модуль был установлен в каталоге `/home/rnd/lib/python` (то есть в домашнем каталоге пользователя `rnd`), а во втором — в каталоге `/usr/local/lib/python2.1/site-packages`.

Если вы работаете в системе Linux, поддерживающей RPM (RedHat Package Manager, менеджер пакетов от RedHat), вы можете поступить более разумно, создав с помощью Distutils RPM-пакет и установив его по обычной для этой системе процедуре. Следующий пример показывает последовательность команд, которая приводит к созданию и установке RPM-пакета (все происходит в одном и том же каталоге):

```
$ python setup.py bdist --formats=rpm
$ su
# rpm -i dist/Combinatorial-1.0-1.noarch.rpm
```

Этот подход хорош тем, что использует системные возможности учета устанавливаемого программного обеспечения. Например, по любому файлу можно легко узнать, к какому пакету он принадлежит, и удалить этот пакет в случае необходимости. В приведенном ниже примере это проделывается для пакета `PyChecker`:

```
$ rpm -qf /usr/lib/python2.1/site-packages/pychecker/checker.py
PyChecker-0.8-1
```

```
$ su
# rpm -e PyChecker
```

Еще одно преимущество UNIX состоит в том, что там почти всегда имеется компилятор языка C, необходимый для компиляции модулей-расширений. Пакетам `Combinatorial` просто "повезло": они состоят из модулей, написанных только на Python.

А вот пакет `Numeric`, например, состоит не только из модулей на Python, но имеет расширения, написанные на C, поэтому для его сборки требуется компилятор. К счастью, такой всегда есть в правильно установленной UNIX-системе (см. список компиляторов в табл. 9.160), и мы можем смело производить установку. Компилятор C для Windows есть на прилагаемом к книге компакт-диске.

Замечание

Правда, пакет `Numeric` состоит из нескольких пакетов, и потому имеет свои индивидуальные особенности. Например, вместо `setup.py` разработчики `Numeric` (версии 20.1) предлагают файл setup_all.py, который и вызывает `setup.py` для всех своих пакетов с теми же опциями, что были заданы `setup_all.py`.

9.22.2. Распространение модулей

Для создания дистрибутива модуля (набора модулей или пакета) нужно выполнить следующее:

- ❒ написать сценарий настройки `setup.py`;
- ❒ написать файл конфигурации настройки `setup.cfg` (необязательно);
- ❒ создать дистрибутив с исходным кодом;
- ❒ построить один или несколько бинарных дистрибутивов (необязательно).

Далее мы рассмотрим эти пункты подробнее.

Сценарий настройки

Сценарий настройки (setup script) — основная забота разработчика при подготовке дистрибутива с помощью Distutils. Этот сценарий является описанием входящих в дистрибутив модулей и пакетов.

В качестве примера мы рассмотрим создание дистрибутива для модуля `Combinatorial` *(см. разд. 4.4)*.

Листинг 9.102. Файл setup.py. Сценарий настройки модуля `Combinatorial`

```
#!/usr/bin/env python
from distutils.core import setup
setup(name="Combinatorial",                    # название дистрибутива
```

```
      version="1.0",                        # версия
      description="Module Combinatorial",   # описание
      author="Roman Suzi",                  # автор
      author_email="rnd@onego.ru",          # email автора
      py_modules=["Combinatorial",],        # состав из модулей
     )
```

Положим этот файл в отдельный каталог, например, с именем `foo`. Туда же положим файл Combinatorial.py:

```
foo
|-- Combinatorial.py
`-- setup.py
```

Файл конфигурации

Помимо сценария настройки `setup.py`, настроечную информацию можно хранить в файле конфигурации. По формату этот файл напоминает INI-файлы, использующиеся в системе Windows:

```
[команда]
опция1=значение1
опция2=многострочное
       значение для
       опции2
...
```

где `команда` — команда, к которой относится опция; `опция` — опция, которой может сопровождаться команда; `значение` — значение опции. Для опций, значения которых в командной строке не изменяются, сам факт их присутствия обозначается значением `1`.

Список команд и опций можно уточнить в *разд. 9.22.3*, а в листинге 9.103 приведен пример файла конфигурации с опциями для построения RPM- и Windows-дистрибутивов:

Листинг 9.103. Файл setup.cfg. Пример файла конфигурации

```
[bdist_rpm]
python=python2.1
group=Misc
release=1
packager="Roman Suzi <rnd@onego.ru>"
vendor="Roman Suzi <rnd@onego.ru>"
[bdist_wininst]
```

```
title=Combinatorial
```

Конфигурационные файлы могут храниться не только в корневом каталоге дистрибутива (локальная конфигурация, относящаяся к конкретному дистрибутиву).

В системе UNIX/Linux-конфигурации различного уровня могут храниться в следующих файлах:

- Системная: `префикс/lib/python-версия/distutils/pydistutils.cfg`
- Пользовательская: `$HOME/.pydistutils.cfg`
- Локальная: `setup.cfg`

В Windows:

- Системная: `префикс\Lib\distutils\pydistutils.cfg`
- Пользовательская: `%HOME\pydistutils.cfg`
- Локальная: `setup.cfg`

На Macintosh:

- Системная: `префикс:Lib:distutils:pydistutils.cfg`
- Пользовательская: (нет)
- Локальная: `setup.cfg`

В приведенных выше полных именах файлов следует заменить `префикс` на конкретный префикс к каталогу, куда установлен Python, `версия` — на используемую версию. Кроме того, переменная окружения `HOME` позволяет пользователю иметь и свою конфигурацию. Если одна и та же опция задана в нескольких местах сразу, отдается предпочтение источнику с бóльшим приоритетом. В порядке возрастания приоритета источники выстраиваются следующим образом:

- значения по умолчанию
- системная конфигурация
- пользовательская конфигурация
- локальная конфигурация
- опции в командной строке

Разумеется, в пользовательской и системной конфигурации имеет смысл указывать опции, не привязанные к конкретному дистрибутиву.

Создание дистрибутивов

Наконец, мы подошли к самому интересному — созданию собственного дистрибутива. Итак, с помощью команд `sdist` и `bdist` можно создать нужные архивы с дистрибутивами:

```
python setup.py sdist --formats=zip,gztar
python setup.py bdist --formats=rpm,wininst
python setup.py clean --all
```

В результате в каталоге `foo` появляется следующее:

```
foo
|-- dist
|   |-- Combinatorial-1.0-1.noarch.rpm
|   |-- Combinatorial-1.0-1.src.rpm
|   |-- Combinatorial-1.0.tar.gz
|   |-- Combinatorial-1.0.win32.exe
|   `-- Combinatorial-1.0.zip
|-- Combinatorial.py
|-- MANIFEST
|-- setup.cfg
`-- setup.py
```

Теперь, если вы пользуетесь Windows, можно запустить программу Combinatorial-1.0.win32.exe, которая и установит модуль `Combinatorial` в вашей системе. В Linux, поддерживающем RPM, установить `Combinatorial` можно по команде:

```
# rpm -hiv Combinatorial-1.0-1.noarch.rpm
```

Файл MANIFEST содержит:

```
Combinatorial.py
setup.cfg
setup.py
```

То есть все те файлы, которые относятся к дистрибутиву с исходными кодами.

9.22.3. Список команд и опций

Утилиты Distutils запускаются в работу путем вызова на исполнение сценария настройки:

- `setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]`

где `global_opts` — глобальные опции; `cmd1`, `cmd2`, ... — команды; `cmd1_opts`, `cmd2_opts`, ... — опции к командам.

В табл. 9.158 приведен список команд Distutils.

Таблица 9.158. *Команды Distutils*

Команда	Описание
`build`	Собирает все необходимое для установки
`build_py`	Копирует модули, написанные на Python, в каталог `build`, где производится сборка
`build_ext`	Собирает (компилирует, компонует) расширения, написанные на C/C++
`build_clib`	Собирает библиотеки на C/C++, используемые в расширениях
`build_scripts`	Копирует сценарии на Python в каталог `build`, изменив если требуется `#!`-строку
`clean`	Очищает временные файлы, оставшиеся после команды `build`
`install`	Устанавливает все из каталога `build`
`install_lib`	Устанавливает все модули (и написанные на Python, и расширения)
`install_headers`	Устанавливает заголовочные файлы C/C++

Таблица 9.158 *(окончание)*

Команда	Описание
`install_scripts`	Устанавливает сценарии
`install_data`	Устанавливает файлы с данными
`sdist`	Создает дистрибутив исходного кода в одном из архивных форматов
`bdist`	Создает собранный бинарный дистрибутив
`bdist_dumb`	Создает простой бинарный дистрибутив (UNIX)
`bdist_rpm`	Создает дистрибутив в виде RPM-файла
`bdist_wininst`	Создает исполняемый файл-установщик для Windows

В табл. 9.159 приведены опции к различным командам. Эти опции можно употреблять в файле конфигурации в разделе для соответствующих команд. Для использования опции в командной строке необходимо предварить ее двумя дефисами `"--"`, например:

```
setup.py --help
```

Кроме того, многие опции требуют указания значения (в таблице к таким опциям добавляют `=x`), например:

```
setup.py build --build-temp=tmp
```

Таблица 9.159. *Опции к различным командам*

Опция	Сокращение	Команды	Описание
`help`	`-h`	Глобальная	Выдает детальную помощь
`verbose`	`-v`	Глобальная	Выдает детальную информацию (по умолчанию)
`quiet`	`-q`	Глобальная	Подавляет выдачу информации о ходе работы
`dry-run`	`-n`	Глобальная	Тестовый прогон. Выдает только сообщения
`build-base=x`	`-b x`	`build`, `clean`	Базовый каталог для сборки библиотек
`build-purelib=x`		`build`	Каталог для сборки платформно-независимых дистрибутивов
`build-platlib=x`		`build`	Каталог для сборки платформно-независимых дистрибутивов

Таблица 9.159 *(продолжение)*

Опция	Сокращение	Команды	Описание
`build-lib=x`		`build`, `clean`	Каталог для сборки дистрибутива
`build-scripts=x`		`build`, `clean`	Каталог для сборки сценариев
`build-temp=x`	`-t x`	`build`, `clean`	Каталог для временных файлов
`compiler=x`	`-c`	`build`	Тип компилятора
`debug`	`-g`	`build`	Компилирует модули-расширения и библиотеки вместе с отладочной информацией
`force`	`-f`	`build`	Собирает все заново, не обращая внимание на времена создания файлов
`help-compiler`		`build`	Список доступных компиляторов. (См. табл. 9.160.)

build-dir=x	-d x	build_scripts	Каталог для сборки
build-clib=x	-b x	build_clib	Каталог для сборки библиотек
build-lib=x	-b x	build_ext	Каталог для сборки модулей расширения
inplace	-i	build_ext	Собирать модули на месте, не копируя в build_lib
include-dirs=x	-I x	build_ext	Список каталогов для поиска заголовочных файлов, через разделитель ":"
define=x	-D	build_ext	Определяет макрос x для препроцессора C
undef=x	-U	build_ext	Снимает определение макроса x
libraries=x	-l	build_ext	Внешние C-библиотеки, с которыми необходимо компоновать модули расширений
library-dirs=x	-L	build_ext	Список каталогов для поиска внешних C-библиотек, через ":"
rpath=x	-R	build_ext	Список каталогов для поиска разделяемых (shared) C-библиотек во время исполнения
link-objects=x	-O	build_ext	Дополнительные объекты, требуемые при компоновке

Таблица 9.159 *(продолжение)*

Опция	Сокращение	Команды	Описание
swig-cpp		build_ext	Заставляет SWIG генерировать файлы на C++ (по умолчанию — C)
compile	-c	build_py, install	Компилировать .py-файлы в .pyc-файлы (для install по умолчанию)
no-compile		build_py, install	Не компилировать .py-файлы (для build_py по умолчанию)
optimize=x	-O x	build_py, install	Компилировать с оптимизацией: x=1 для "python -O", x=2 для "python -OO" и x=0 (по умолчанию) — без оптимизации
all	-a	clean	Удалить все результаты сборки, а не только временные файлы

prefix=x	install	Префикс установки
exec-prefix=x	install	Префикс для платформно-зависимых файлов (UNIX)
home=x	install	Домашний каталог, в который произвести установку (UNIX)
install-base=x	install	Базовый каталог для установки (вместо prefix и home)
install-platbase=x	install	Базовый каталог для платформно-зависимых файлов (вместо exec-prefix и home)
root=x	install	Устанавливать относительно заданного альтернативного корня (например, если установка происходит на сетевой диск)
install-purelib=x	install	Каталог для модулей на Python
install-platlib=x	install	Каталог для установки дистрибутивов с модулями-расширениями
install-lib=x	install	Каталог для установки (вместо install-purelib и install-platlib)
install-headers=x	install	Каталог для установки заголовочных файлов для C/C++
install-scripts=x	install	Каталог для установки сценариев

Таблица 9.159 *(окончание)*

Опция	Сокращение	Команды	Описание
record=x		install	Задает файл для записи списка устанавливаемых файлов
force	-f	install	В ходе установки удаляет существующие файлы
install-dir=x	-d x	install_lib, install_scripts, install_data, install_headers	Каталог для установки
build-dir=x	-b x	install_lib	Каталог для сборки
template=x	-t x	sdist	Имя шаблона для перечня файлов ди-

			стрибутива (по умолчанию MANIFEST.in)
manifest=x	-m	sdist	Имя файла с перечнем файлов (по умолчанию: MANIFEST)
use-defaults		sdist	Включать в перечень набор файлов по умолчанию
no-defaults		sdist	Не включать в перечень набор файлов по умолчанию
prune		sdist	Исключать из дистрибутива некоторые известные файлы и каталоги (build, RCS, VCS и т. п.)
no-prune		sdist	Отменяет prune
manifest-only	-o	sdist	Только регенерировать перечень файлов
force-manifest	-f	sdist	Насильно регенерировать перечень файлов и продолжить работу как обычно
formats=x		sdist, bdist	Форматы, в которых следует сделать дистрибутив, перечисленные через запятую
dist-dir=x	-d x	sdist, bdist	Каталог для записи архивов с дистрибутивами (по умолчанию: dist)
help-formats		sdist, bdist	Помощь по возможным форматам архивов (табл. 9.161)
bdist-base=x	-b x	bdist	Временный каталог для создания бинарных дистрибутивов

Примечание

Некоторые опции работают не только с указанными родовыми командами типа install, но и с их специальными аналогами. Например, опция no-compile будет работать как с командой install, так и с install_lib. Применимость той или иной опции можно узнать с помощью опции --help (к сценарию setup.py), за которой следует имя команды. Кроме того, не указаны слишком специфичные опции для команд bdist_rpm и bdist_wininst.

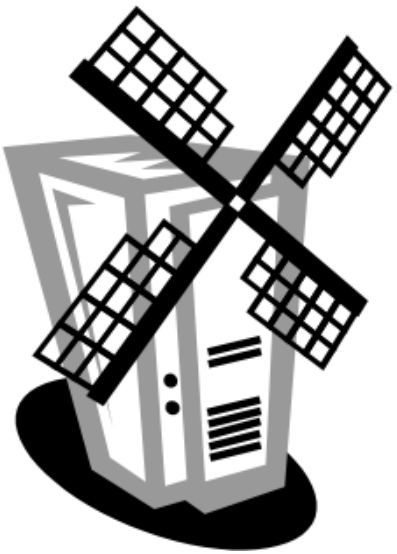
Таблица 9.160. *Список доступных компиляторов*

Опция `compiler`	**Описание**
bcpp	Borland C++
cygwin	Cygwin, аналог GNU C для Win
mingw32	Mingw32, аналог GNU C для Win32 (имеется на ком-

	пакт-диске)
`Msvc`	Microsoft Visual C++
`Mwerks`	MetroWerks CodeWarrior
`Unix`	Стандартный компилятор в UNIX

Таблица 9.161. *Форматы дистрибутивов*

Опция `formats`	**Описание**
`Bztar`	tar-архив, сжатый `bzip2`
`Gztar`	tar-архив, сжатый `gzip`
`Tar`	tar-архив
`Zip`	ZIP-файл
`Ztar`	tar-архив, сжатый `compress`
`Rpm`	RPM-файл (для бинарных дистрибутивов)
`Wininst`	Исполняемый установщик для Windows (для бинарных дистрибутивов)



Часть III

Расширения Python

Эта часть дает общее представление о некоторых наиболее популярных расширениях языка Python для различных приложений, а также об интеграции с другими средами программирования.

Глава 10. Python и C

Глава 11. Numeric Python

Глава 12. Обработка изображений: Python Image Library

Глава 13. Связь с базами данных

Глава 14. Среда Zope

Глава 15. Jython — Python на Java

Глава 10

Python и C



Интерпретатор языка Python можно как расширить за счет *модулей расширения (extension modules)*, так и встроить (embed) в другие программы. Модули расширения можно писать на языке C или C++ и вызывать из программ на Python. И наоборот, из программ на C можно вызывать интерпретатор Python.

Необходимость использования языка C может возникнуть, если соответствующие действия требуют бóльшей производительности, чем позволяет Python. Например, высокопроизводительные операции с массивами модуля Numeric *(см. гл. 11)* написаны на C. Это позволяет объединить эффективность C-компилятора с удобством и гибкостью интерпретатора Python. Создание модулей расширения для Python хорошо описано в документе "Python/C API", входящем в состав стандартной документации. В этой главе на конкретном примере будет продемонстрировано, как можно написать собственный модуль расширения.

В качестве примера мы рассмотрим модуль с функцией для подсчета количества различных символов в строке. На Python требуемая функция имеет следующий вид:

```
def frequency(s):
  freq = {}
  for c in s:
    freq[c] = (freq.get(c) or 0) + 1
  return freq
```

На языке C похожая функция будет выглядеть так:

```
#include <stdlib.h>

int * frequency(char s[]) {
  int *freq;
  char *ptr;
  freq = (int*)(calloc(256, sizeof(int)));
  if (freq != NULL)
    for (ptr = s; *ptr; ptr++)
```

```
      freq[*ptr] += 1;
  return freq;
}
```

Заметьте, что функция выделения динамической памяти `calloc()` заодно и заполняет область памяти нулями, поэтому нам не требуется делать это отдельно.

Рассмотрим теперь как создать модуль расширения так, чтобы из Python можно было вызвать функцию `frequency()` этого модуля.

Прежде всего, любой модуль расширения должен включать заголовочный файл Python.h. Этот заголовочный файл поставляется вместе с Python, и компилятору C должен быть указан путь к каталогу, где он находится. Этот же заголовочный файл может использоваться и компилятором C++. Настоятельно рекомендуется не использовать имен, начинающихся с `Py` или `_Py`, т. к. они могут конфликтовать с именами, определенными в Python.h.

Для обозначения объекта языка Python в Python.h определен тип `PyObject`, поэтому функции, которые будут вызываться из интерпретатора Python, должны принимать и возвращать указатели этого типа. За исключением особого типа `PyTypeObject` (объект-тип), под объекты Python память, как правило, выделяется динамически. О своих объектах Python позаботится сам, однако, если в программе на языке C делаются какие-то дополнительные ссылки на `PyObject`, там же должны использоваться макросы `Py_INCREF` и `Py_DECREF`, которые, соответственно, увеличивают и уменьшают счетчик ссылок на конкретный объект. Если не увеличивать количество ссылок, программа на C может столкнуться с тем, что сохраненный в ней указатель соответствует области памяти, в котором уже нет старого объекта. Это очень опасно и может вызвать совершенно непредсказуемые ошибки. С другой стороны, неиспользование `Py_DECREF` может повлечь утечку памяти: объекты не будут удаляться из-за завышенного значения счетчика ссылок.

Для облегчения задачи функции вроде `PyObject_()`, `PyNumber_()`, `PySequence_()`, `PyMapping_()` автоматически увеличивают счетчик ссылок на этот объект. (`Py_DECREF` остается, однако, на совести разработчика.)

В листинге 10.1 приведен текст модуля на языке C, реализующего функцию `frequence()`. Модуль назван `freq`, поэтому по сложившейся традиции C-файл должен быть назван freqmodule.c.

Листинг 10.1. Файл freqmodule.c. Подсчет частоты встречаемости символов

```
#include <Python.h>
#include <stdlib.h>
```

```
/* C-функция в неизменном виде */
long * frequency(char s[]) {
  long *freq;
  char *ptr;
  freq = (long*)(calloc(256, sizeof(long)));
  if (freq != NULL)
    for (ptr = s; *ptr; ptr++)
      freq[*ptr] += 1;
  return freq;
}

/* обертка для вызова из Python */
static PyObject * freq_frequency(PyObject *self, PyObject *args) {
  char *src;
  long *res;
  int i;
  PyObject *result;

  /* переводим (первый и единственный) аргумент в строку C */
  if (!PyArg_ParseTuple(args, "s:frequency", &src))
    return NULL;

  /* подсчитываем частоты */
  res = frequency(src);

  /* формируем кортеж с результатом */
  result = PyTuple_New(256);
  for(i=0; i<256; i++)
    PyTuple_SetItem(result, i, PyLong_FromLong(res[i]));

  /* освобождаем память из-под res */
  free(res);

  /* возвращаем результат */
  return result;
}
```

```
/* массив с функциями модуля */
static PyMethodDef freq_methods[] = {
  {"frequency", freq_frequency,   METH_VARARGS,
   "frequency(s) -> tuple_of_frequencies\n\
  Return frequencies of chars encountered in string s as a tuple."},
  {NULL, NULL}           /* that's all */
};

/* строка документации */
static char freq__doc__[] =
 "Module for calculating some statisctics.\n";

/* инициализация модуля */
DL_EXPORT(void)
initfreq(void) {
   Py_InitModule3("freq", freq_methods, freq__doc__);
}
```

Приведенный в листинге 10.1 модуль рассчитан на использование в виде динамически подгружаемой библиотеки. Чтобы его скомпилировать, в системе с компилятором GNU C достаточно подать следующие команды:

```
$ gcc -g -O2 -I/usr/include/python2.1 -c freqmodule.c -o freqmodule.o
$ gcc -shared freqmodule.o -o freq.so
```

При использовании Microsoft Visual C++ команда будет выглядеть примерно так:

```
cl /LD /I/Python21/include freq.c /Python21/libs/python21.lib
```

Если теперь запустить Python в одном каталоге с файлом freq.so (или freq.dll), мы сможем использовать созданный модуль:

```
>>> import freq
>>> freq.frequency("ABCDE"*10+"FG"*20)[60:75]
(0L, 0L, 0L, 0L, 0L, 10L, 10L, 10L, 10L, 10L, 20L, 20L, 0L, 0L, 0L)
```

Разумеется, лучше всего компилировать модуль расширения тем же компилятором, которым был скомпилирован Python. Добиться нужного результата легче, если воспользоваться `Distutils` *(см. разд. 9.22)*. Следующий файл setup.py позволит создать дистрибутив для модуля `freq`:

```
from distutils.core import setup, Extension

setup(name = "freq",
      version = "0.1",
```

```
    maintainer = "me",
    maintainer_email = "my@email",
    description = "Some statisctics",
    url = "not yet",
    ext_modules = [Extension('freq', ['freqmodule.c',]), ]
  )
```

Если в вашей системе все правильно настроено, следующая команда откомпилирует модуль и создаст бинарный дистрибутив:

```
python setup.py bdist
```

Главное, на что следует обратить внимание при создании модуля расширения, — управление памятью. Заметьте, что в нашей программе не были использованы `Py_INCREF` или `Py_DECREF`. В случае, когда функции `PyTuple_New()` и `PyLong_FromLong()` выполняются без ошибок, ничего страшного не случится. Однако, если в этих функциях будут ошибки, программа (и интерпретатор), скорее всего, завершатся аварийно из-за некорректного доступа к памяти. Чтобы этого не случилось, необходимо проверять результаты, возвращаемые функциями `frequency()`, `PyTuple_New()` и `PyLong_FromLong()`. (Кстати, функция `frequency()` устроена так, что возвращает `NULL`, если ей не удалось выделить память.)

Листинг 10.2. C-код функции `freq.frequency`

```
/* обертка для вызова из Python */
static PyObject * freq_frequency(PyObject *self, PyObject *args) {
  char *src;
  long *res;
  int i;
  PyObject *result, *long_item;

  /* переводим (первый и единственный) аргумент в строку C */
  if (!PyArg_ParseTuple(args, "s:frequency", &src))
    return NULL;

  /* подсчитываем частоты */
  res = frequency(src);
  if (res == NULL)
    goto memory_error;
```

```
  /* формируем кортеж с результатом */
  result = PyTuple_New(256);
  if (result == NULL)
    goto memory_error;

  for(i=0; i<256; i++) {
    long_item = PyLong_FromLong(res[i]);
    if (long_item == NULL) {
      for(i--; i>=0; i--)
        Py_DECREF(PyTuple_GetItem(result, i));
      Py_DECREF(result);
      goto memory_error;
    }
    PyTuple_SetItem(result, i, long_item);
  }

  /* освобождаем память из-под res */
  free(res);

  /* возвращаем результат */
  return result;

  /* возбуждаем исключение MemoryError */
memory_error:
  PyErr_SetString(PyExc_MemoryError, "Out of memory.");
  return NULL;
}
```

Вот теперь наша функция будет выполняться корректно, возбуждая `MemoryError` в случае невозможности выделения памяти.

Модуль расширения Python может гораздо больше, чем мы смогли здесь рассмотреть. Кратко можно сказать так: API содержит абсолютно все для того, чтобы в модуле-расширении можно было обращаться к объектам из среды Python. И даже больше того: можно создавать собственные типы данных, которые будут работать подобно встроенным. Конечно, необходимо отдавать себе отчет в том, что некорректно написанный модуль может подорвать стабильность всей программы, включая интерпретатор Python.

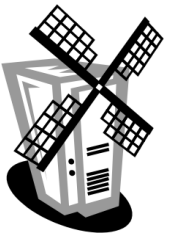
Предупреждение

В Python 2.2 C API может измениться, т. к., начиная с этой версии, встроенные типы данных будут мало чем отличаться от классов, определенных пользователем. Различие между типами и классами будет стерто, что наверняка повлечет изменения в C API.

Встраивание в программу на C или C++ интерпретатора Python — дело несколько более трудоемкое, т. к. требует исходных текстов Python и библиотек, которые образуются при компиляции Python. В каталоге `Demo/embed` дистрибутива исходных кодов Python имеются примеры программ, которые используют встроенный интерпретатор.

Глава 11

Numeric Python



Numeric Python (численный Python) — собрание модулей, в центре которого быстродействующее ядро для работы с многомерными массивами, необходимое для многих численных приложений. В какой-то степени Numeric вносит в Python возможности таких пакетов и систем, как MatLab, Octave (аналог MatLab), APL, J, S+, IDL. Пользователи этих пакетов без труда обнаружат в Numeric знакомые черты.

Numeric Python имеет средства для:

- матричных вычислений (`LinearAlgebra`)
- быстрого преобразования Фурье (`FFT`)
- работы с недостающими экспериментальными данными (`MA`)
- статистического моделирования (`RNG`)

В необходимых случаях Numeric может быть сконфигурирован для работы со ставшими классическими численными библиотеками `LAPACK` и `FFTPACK` (в их отсутствие используется облегченная версия `LAPACK` — `LALITE`). Кроме того, имеется специальный модуль `MLab` для эмуляции базовых функций программы MatLab.

Ниже будут коротко рассмотрены возможности модулей `Numeric` и `LinearAlgebra`.

11.1. Модуль *Numeric*

Модуль `Numeric` вносит в Python полноценный объект-массив и большое число функций для оперирования с ним.

Массив — набор однородных элементов. По сравнению с массивами модуля `array`, массивы модуля `Numeric` могут иметь более одной *размерности*.

Массив можно создать с помощью функции `array()`:

```
from Numeric import *
print array([[1, 2], [3, 4], [5, 6]])
print array([[1, 2, 3], [4, 5, 6]], Float)
print array([65, 66, 67, 68, 69], 'c')
```

Результат выполнения:

```
[[1 2]
 [3 4]
 [5 6]]
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
[A B C D E]
```

Для элементов массивов можно использовать *типы* `Int8`—`Int32`, `UnsignedInt8`, `Float8`—`Float64`, `Complex8`—`Complex64` и `PyObject`. Числа 8, 16, 32 и 64 обозначают количество бит для хранения величины. Типы `Int`, `Float` и `Complex` соответствуют наибольшим принятым на данной платформе значениям. В массивах можно также хранить ссылки на произвольные объекты (тип `PyObject`), но это, естественно, будет сказываться на производительности. Для некоторых применений операции, существующие для числовых массивов, очень удобны и для массивов с произвольными объектами.

Количество размерностей (осей) и длина массива по каждой оси называются *формой (shape) массива*. Прочитать или задать форму можно с помощью атрибута `shape`:

```
from Numeric import *
a = array(range(12), Int)
print "a.shape:", a.shape
print a
a.shape = (3, 4)
print "a.shape:", a.shape
print a
```

Результат выполнения:

```
a.shape: (12,)
[ 0  1  2  3  4  5  6  7  8  9 10 11]
a.shape: (3, 4)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Для задания формы можно использовать и функцию `Numeric.reshape()`, причем, действовать она может сразу над последовательностями, создавая объект-массив.

```
>>> import Numeric
>>> print Numeric.reshape("правда", (2, -1))
[[п р а]
 [в д а]]
```

В этом примере `-1` в указании формы — не случайность. Общее количество элементов массива известно (в примере — `6`), поэтому длину одной (и не более одной!) из осей можно не задавать, записывая на ее месте `-1`.

Атрибут `flat` выпрямляет массив в одномерный:

```
>>> import Numeric
>>> a = array([[1, 2], [3, 4]])
>>> print a.flat
[1 2 3 4]
```

Похожа на `Numeric.reshape()` функция `Numeric.resize()` для придания массиву необходимой формы и размера:

```
>>> import Numeric
>>> print Numeric.resize("правда", (10,))
[п р а в д а п р а в]
```

Как уже говорилось при рассмотрении объектов-срезов, объекты-массивы `Numeric` используют расширенный синтаксис выделения среза. Следующие наборы примеров хорошо иллюстрируют различные виды записи срезов. Здесь же мы видим функцию `Numeric.arrayrange()`, которая является аналогом `range()` для массивов.

```
import Numeric
a = Numeric.arrayrange(24) + 1  # массив чисел от 1 до 24
a.shape = (4, 6)                # форма 4x6
examples = [ ("a[1,2]", "элемент 1,2"),
             ("a[1,:]", "строка 1"),
             ("a[1]", "тоже строка 1"),
             ("a[:,1]", "столбец 1"),
             ("a[-2,:]", "предпоследняя строка"),
             ("a[0:2,1:3]", "окно 2x2"),
             ("a[1,::3]", "каждый третий элемент из строки 1"),
             ("a[:,::-1]", "реверс элементов в каждой строке"),
             ]
print a
for (expr, expl) in examples:
  print "%s (%s):\n%s" % (expr, expl, str(eval(expr)))
```

Результат выполнения:

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
```

```
a[1,2] (элемент 1,2):
9
a[1,:] (строка 1):
[ 7  8  9 10 11 12]
a[1] (тоже строка 1):
[ 7  8  9 10 11 12]
a[:,1] (столбец 1):
[ 2  8 14 20]
a[-2,:] (предпоследняя строка):
[13 14 15 16 17 18]
a[0:2,1:3] (окно 2x2):
[[2 3]
 [8 9]]
a[1,::3] (каждый третий элемент из строки 1):
[ 7 10]
a[:,::-1] (реверс элементов в каждой строке):
[[ 6  5  4  3  2  1]
 [12 11 10  9  8  7]
 [18 17 16 15 14 13]
 [24 23 22 21 20 19]]
```

Второй набор примеров использует многоточие (`Ellipsis`). Многоточие ставится для указания произвольного числа осей, т. е. вместо `:,:,...,:`.

```
import Numeric
a = Numeric.arrayrange(24) + 1  # массив чисел от 1 до 24
a.shape = (2,2,2,3)             # форма 2x2x2x3
examples = [ ("a[0,...]",   "0-й блок"),
             ("a[0,:,:,0]", "срез по первой и последней осям"),
             ("a[0,...,0]", "то же"),
           ]
print a
for (expr, expl) in examples:
  print "%s (%s):\n%s" % (expr, expl, str(eval(expr)))
```

Результат выполнения:

```
[[[[ 1  2  3]
   [ 4  5  6]]
  [[ 7  8  9]
   [10 11 12]]]
```

```
 [[[13 14 15]
   [16 17 18]]
  [[19 20 21]
   [22 23 24]]]]
a[0,...] (0-й блок):
[[[ 1  2  3]
  [ 4  5  6]]
 [[ 7  8  9]
  [10 11 12]]]
a[0,:,:,0] (срез по первой и последней осям):
[[ 1  4]
 [ 7 10]]
a[0,...,0] (то же):
[[ 1  4]
 [ 7 10]]
```

Для порождения массивов, состоящих из одних нулей или единиц, можно использовать функции `zeros()` и `ones()` соответственно. Для порождения *единичной матрицы* размера `n` можно воспользоваться `identity(n)`.

Отдельно требуется затронуть вопрос о *копировании массивов*. Дело в том, что во многих операциях, которые только меняют форму или выделяют некоторые срезы массива, результат представляет лишь еще одно "окно" к тем же данным, и изменение данных в этом окне сказывается на исходном массиве. Следующий пример показывает, что для копирования массивов лучше явно использовать `copy()` из модуля `copy`.

```
from Numeric import *
from copy import copy
a = array([1, 2, 3, 4, 5, 6, 7, 8])
print a
b, c, d = a[:2], reshape(a, (4, 2)), copy(a)
b[0], c[0, 1], d[2] = -1, -2, -3
print a
```

Результат выполнения:

```
[1 2 3 4 5 6 7 8]
[-1 -2  3  4  5  6  7  8]
```

Функция `array()` для создания массива также делает копию, если ее аргумент — массив. Кроме того, имеется функция `asarray()`, которая не создает нового массива, когда ее аргумент уже массив.

Модуль `Numeric` определяет целый набор функций, аналогичных соответствующим встроенным функциям, функциям из модуля `math` и др. В табл. 11.1 приведен список доступных функций, называемых универсальными в силу того, что они применимы не только к массивам, но и к последовательностям (в результате все равно получается массив). *См. также разд. 9.4.1*, где перечислены функции модуля `math`.

Таблица 11.1. *Универсальные функции модуля Numeric*

Название функции	Действие функции
`add()`, `subtract()`	Сложение, вычитание
`multiply()`, `divide()`, `remainder()`, `fmod()`	Умножение, деление, получение остатка от деления (для целых чисел и чисел с плавающей запятой)
`power()`	Возведение в степень
`sqrt()`	Корень квадратный
`negative()`, `absolute()`, `fabs()`	Смена знака и абсолютное значение
`ceil()`, `floor()`	Наименьшее (наибольшее) целое, большее (меньшее) или равное аргументу
`hypot()`	Длина гипотенузы (даны два катета)
`sin()`, `cos()`, `tan()`	Тригонометрические функции
`arcsin()`, `arccos()`, `arctan()`	Обратные тригонометрические функции
`arctan2()`	Арктангенс от частного аргументов
`sinh()`, `cosh()`, `tanh()`	Гиперболические функции
`arcsinh()`, `arccosh()`, `arctanh()`	Обратные гиперболические функции
`exp()`	Экспонента (e в степени)
`log()`, `log10()`	Натуральный и десятичный логарифмы
`maximum()`, `minimum()`	Максимум и минимум
`conjugate()`	Сопряжение (для комплексных чисел)
`equal()`, `not_equal()`	Равно, не равно
`greater()`, `greater_equal()`	Больше, больше или равно
`less()`, `less_equal()`	Меньше, меньше или равно
`logical_and()`, `logical_or()`, `logical_xor()`, `logical_not()`	Логические И, ИЛИ, исключающее ИЛИ и отрицание

Таблица 11.1 *(окончание)*

Название функции	Действие функции
`bitwise_and()`, `bitwise_or()`, `bitwise_xor()`, `invert()`	Побитовые И, ИЛИ, исключающее ИЛИ и инверсия
`left_shift()`, `right_shift()`	Побитовые сдвиги влево и вправо

Эти функции являются объектами типа `ufunc` и применяются к массивам поэлементно. Кроме того, их специальные методы `accumulate`, `outer`, `reduce` и `reduceat` предназначены для выполнения операций над массивами в определенном режиме (соответственно): аккумулирование результата, внешнее "произведение", сокращение и сокращение с заданными точками.

В представленных функциях и их применении над массивами в различных режимах кроется огромный потенциал. С помощью умелого подбора нужной комбинации операций можно запрограммировать весьма нетривиальные алгоритмы, которые будут выполняться гораздо быстрее, чем их аналоги, записанные с применением вложенных циклов. Однако проще всего объяснить действие функции на примере сложения `add`.

```
>>> from Numeric import add
>>> add([[1, 2],
...      [3, 4]], [[1, 0],
...                [0, 1]])
[[2 2]
 [3 5]]
>>> add([[1, 2],
...      [3, 4]], [1, 0])
[[2 2]
 [4 4]]
>>> add([[1, 2], [3, 4]], 1)
[[2 3]
 [4 5]]
>>> add.reduce([1,2,3,4])            # т. е. 1+2+3+4
10
>>> add.reduce([[1, 2],              # т. е. [1+3 2+4]
...             [3, 4]], 0)
[4 6]
>>> add.reduce([[1, 2],              # т. е. [1+2 3+4]
...             [3, 4]], 1)
[3 7]
```

```
>>> add.accumulate([1,2,3,4])        # т. е. [1 1+2 1+2+3 1+2+3+4]
[ 1  3  6 10]
>>> add.reduceat(range(10),[0,3,6])  # т. е. [0+1+2 3+4+5 6+7+8+9]
[ 3 12 30]
>>> add.outer([1,2], [3,4])          # т. е. [[1+3 1+4] [2+3 2+4]]
[[4 5]
 [5 6]]
```

Методы `accumulate()`, `reduce()` и `reduceat()` принимают третий, необязательный, аргумент — номер оси, которая используется для соответствующего действия. По умолчанию используется нулевая ось.

Сами универсальные функции помимо одного или двух необходимых аргументов позволяют задавать и еще один аргумент, который принимает результат производимой функцией операции. Тип третьего аргумента должен строго соответствовать типу результата. Например `sin`, взятый от целых чисел, может иметь тип `Float`.

```
>>> from Numeric import array, sin, Float
>>> a = array([0, 1, 2])
>>> r = array([0, 0, 0], Float)
>>> sin(a, r)                        # sin(a) поместить в r
>>> print r
[ 0.          0.84147098  0.90929743]
```

В табл. 11.2 приводятся функции модуля `Numeric`, которые являются краткой записью некоторых наиболее употребительных сочетаний функций и методов из рассмотренных выше.

Таблица 11.2. *Некоторые функции модуля* `Numeric` *и их аналоги*

Функция и ее аргументы	Аналог функции
`sum(a, axis)`	`add.reduce(a, axis)`
`cumsum(a, axis)`	`add.accumulate(a, axis)`
`product(a, axis)`	`multiply.reduce(a, axis)`
`cumproduct(a, axis)`	`multiply.accumulate(a, axis)`
`alltrue(a, axis)`	`logical_and.reduce(a, axis)`
`sometrue(a, axis)`	`logical_or.reduce(a, axis)`

Следующий пример показывает, что всегда следует внимательно смотреть, где размещается результат операции:

Листинг 11.1. Клеточный автомат

```
from Numeric import *
lenta, lenta[5] = zeros(16), 1
print "Пытаемся делать сложение на месте:"
for i in xrange(3):
  print lenta
  add(lenta[:-1], lenta[1:], lenta[1:])

lenta, lenta[5] = zeros(16), 1
print "Теперь используем промежуточный массив:"
for i in xrange(3):
  print lenta
  lenta[1:] = add(lenta[:-1], lenta[1:])

lenta, lenta[5] = zeros(16), 1
a = zeros(15)
print "Используем явный промежуточный массив:"
for i in xrange(3):
  print lenta
  add(lenta[:-1], lenta[1:], a)
  lenta[1:] = a
```

Результат выполнения:

```
Пытаемся делать сложение на месте:
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1]
[ 0  0  0  0  0  1  2  3  4  5  6  7  8  9 10 11]
Теперь используем промежуточный массив:
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 2 1 0 0 0 0 0 0 0 0]
Используем явный промежуточный массив:
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 2 1 0 0 0 0 0 0 0 0]
```

В первом случае мы попытались сэкономить на памяти и получили испорченный результат. Второй и третий варианты избежали ошибки, т. к. результат операции в них сохраняется в промежуточном массиве. К счастью, в нашем случае ошибку легко увидеть, чего не скажешь о сложных матричных алгоритмах.

Для сохранения массивов (и других объектов) в файлоподобных объектах можно либо использовать функции `Numeric.dump()` и `Numeric.load()`, либо универсальные функции модуля `pickle` *(см. разд. 9.2.7)*. Судя по исходным текстам модуля `Numeric`, лучше воспользоваться первым вариантом:

```
from Numeric import *
file = open('my.dat', 'wb')
a1 = array(range(500), Float)
a2 = reshape(range(250), (25, -1))
a3 = {'my' : 'data'}
dump((a1, a2, a3), file)
print a1[100], a2[0, 0], a3
file.close()
# возможно, уже в другой программе, даже на другой машине:
file = open('my.dat', 'rb')
(a, b, c) = load(file)
print a[100], b[0, 0], c
```

Настал черед рассказать о других функциях работы с массивами.

Функция `Numeric.take()` позволяет (дословно) взять часть массива по заданным на определенной оси индексам. По умолчанию номер оси (третий аргумент) равен 0.

```
>>> from Numeric import *
>>> a = reshape(arrayrange(25), (5, 5))
>>> print a
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
>>> print take(a, [1], 0)
[ [5 6 7 8 9]]
>>> print take(a, [1], 1)
[[ 1]
 [ 6]
```

```
 [11]
 [16]
 [21]]
>>> print take(a, [[1,2],[3,4]])
[[[ 5  6  7  8  9]
  [10 11 12 13 14]]
 [[15 16 17 18 19]
  [20 21 22 23 24]]]
```

В отличие от операции выделения среза, функция `Numeric.take()` оставляет размерность массива той же, если конечно, структура заданных индексов одномерна. Результат `take(a, [[1,2],[3,4]])` показывает, что взятые по индексам части помещаются в массив со структурой самих индексов, как если вместо `1` написать `[ 5 6 7 8 9]`, вместо `2` взять `[10 11 12 13 14]` и т. д.

Функция `Numeric.diagonal()` возвращает диагональ.

```
diagonal(a[, смещ[, ось1[, ось2]]]) -> a1
```

где `a` — исходный массив; `смещ` — смещение вправо от "главной" диагонали (по умолчанию 0); `ось1` — первая из осей, на которых берется диагональ (по умолчанию 0); `ось2` — вторая ось (по умолчанию 1); `a1` — массив с результатом.

```
>>> from Numeric import *
>>> a = reshape(arrayrange(16), (4, 4))
>>> print a
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
>>> for i in range(-3, 4):
...   print diagonal(a, i)
...
[12]
[ 8 13]
[ 4  9 14]
[ 0  5 10 15]
[ 1  6 11]
[2 7]
[3]
```

Наверное, к этому моменту идея модуля `Numeric` стала понятна. В табл. 11.3 приведены описания функций модуля.

Таблица 11.3. *Функции модуля* `Numeric`

Функция и ее аргументы	Назначение функции
`array(a[, тип [, savespace]])`	Создание массива на основе последовательности `a` данного `типа`. Аргумент `savespace` является флагом, указывающим необходимость сохранения места
`zeros(форма[, тип])`	Массив нулей заданной формы и типа
`ones(форма[, тип])`	Массив единиц заданной формы и типа
`fromstring(строка [, колво[, тип]])`	Создание массива на основе бинарных данных, хранящихся в строке
`take(a, индексы[, ось])`	Выбор частей массива `a` на основе `индексов` по `оси`
`put(a, индексы, b)`	Присваивание частям массива, `a[n] = b[n]` для всех индексов `n`
`putmask(a, маска, b)`	Присваивание `a` элементов из `b`, для которых маска имеет значение "истина"
`reshape(a, форма)`	Возвращает массив нужной формы (нового массива не создает)
`repeat(a, n[, ось])`	Повторяет элементы массива `a` n раз по `оси`
`choose(a, (b0,...,bn))`	Конструирует массив на основе элементов, взятых по индексам из `a` (индексы от `0` до `n` включительно). Формы массивов `a`, `b1`, ..., `bn` должны совпадать
`cross_correlate(a, b [, режим])`	Взаимная корреляция двух массивов. Параметр `режим` может быть равен `0`, `1` или `2`
`searchsorted(a, i)`	Для каждого элемента из `i` найти его место в массиве `a`. Массив `a` должен быть одномерным и отсортированным. Результат имеет форму массива `i`
`sum(a[, ось])`	Суммирование по оси `ось` массива `a`
`cumsum(a[, ось])`	Суммирование с промежуточными результатами
`product(a[, ось])`	Произведение по оси `ось` массива `a`
`cumproduct(a[, ось])`	Произведение по оси `ось` массива `a` с промежуточными результатами
`alltrue(a[, ось])`	Логическое И по всей оси `ось` массива `a`
`sometrue(a[, ось])`	Логическое ИЛИ по всей оси `ось` массива `a`
`allclose(a, b [, eps[, A]])`	Сравнение `a` и `b` с заданными относительными `eps` и абсолютными `A` погрешностями. По умолчанию `eps` равен `1.0e-1`, а `A` — `1.0e-8`

Таблица 11.3 *(продолжение)*

Функция и ее аргументы	Назначение функции
`arrayrange(start [, stop[, step[, тип]]])` или `arange()`	Аналог `range()` для массивов
`asarray(a[, тип [, savespace]])`	То же, что и `array`, но не создает новый массив, если `a` — уже массив
`convolve(a, b[, режим])`	Свертка двух массивов. Аргумент `режим` может быть равен `0`, `1` или `2`
`swapaxes(a, ось1, ось1)`	Смена осей (частный случай транспонирования)
`concatenate(a[, ось])`	Соединение двух массивов (конкатенация) по заданной `оси` (по умолчанию — по нулевой оси).
`transpose(a[, оси])`	Перестановка осей в соответствии с аргументом `оси`, либо, если `оси` не заданы — расположение их в обратном порядке
`sort(a[, ось])`	Сортировка элементов массива по заданной оси
`argsort(a[, ось])`	Индексы отсортированного массива, такие, что `take(a,argsort(a, axis),axis)` дает отсортированный массив `a`, как если бы делалось `sort(a, axis)`
`argmax(a[, ось])`	Индекс максимального значения в массиве на заданной `оси`
`argmin(a[, ось])`	Индекс минимального значения в массиве на заданной `оси`
`innerproduct(a, b)`	Внутреннее произведение двух массивов (по общему измерению). Для успеха операции `a.shape[-1]` должен быть равен `b.shape[-1]`. Форма результата будет равна `a.shape[:-1] + b.shape[:-1]`. Элементы пропадающего измерения попарно умножаются и суммируются
`dot(a, b)`	Внутреннее (матричное) произведение массивов. По определению: `innerproduct(a, swapaxes(b, -1, -2))`, т. е. с переставленными последними осями, как и подобает произведению матриц
`outerproduct(a, b)`	Внешнее произведение `a` и `b`
`resize(a, форма)`	Возвращает массив с произвольной новой формой
`indices(измерения [, тип])`	Возвращает массив сеток индексов заданной длины по каждому измерению с изменением поочередно по каждому измерению. Например, `indices([2, 2])[1]` дает двумерный массив `[[0, 1], [0, 1]]`

Таблица 11.3 *(продолжение)*

Функция и ее аргументы	Назначение функции
`fromfunction(f, измерения)`	Строит массив путем вызова функции `f()`, в качестве аргументов которой выступают значения кортежа сеток индексов. Фактически является сокращением для `apply(f, tuple(indices(измерения)))`
`diagonal(a[, k[, ось1[, ось2]]])`	Взятие `k`-й диагонали массива `a` в плоскости осей `ось1` и `ось2`
`trace(a[, k[, ось1[, ось2]]])`	Сумма элементов вдоль диагонали. То есть `add.reduce(diagonal(a, k, ось1, ось2))`
`dump(obj, file)`	Запись массива `a` (в двоичном виде) в открытый файлоподобный объект `file`. Файл должен быть открыт в бинарном режиме. Можно записать несколько объектов подряд
`dumps(obj)`	Строка с двоичным представлением объекта `obj`
`load(file)`	Чтение массива из файла (или файлоподобного объекта) `file`. Файл должен быть открыт в бинарном режиме
`loads(s)`	Возвращает объект, соответствующий бинарному представлению в строке
`ravel(a)`	Распрямление массива в одномерный. Аналогично `reshape(a, (-1,))`
`nonzero(a)`	Возвращает индексы ненулевых элементов одномерного массива
`shape(a)`	Возвращает форму массива `a`
`where(массив_условие, м1, м2)`	Выбор элементов на основании `массив_условие` из `м1` (если не ноль) и `м2` (при нуле) поэлементно. Равносилен `choose(not_equal(condition, 0), (y, x))`. Формы массивов-аргументов должны совпадать
`compress(условие, a[, ось])`	Возвращает массив только из тех элементов массива `a`, для которых `условие` истинно (не ноль)
`clip(a, a_min, a_max)`	Обрубает значения массива `a` так, чтобы они находились между значениями из `a_min` и `a_max` поэлементно. Если элемент выходит за пределы, указанные элементами с теми же индексами в `a_min` и `a_max`, он принимает ближайшее значение из этого промежутка
`zeros(форма[, тип])`	Массив из нулей заданной `формы` и `типа`

Таблица 11.3 *(окончание)*

Функция и ее аргументы	Назначение функции
`ones(форма[, тип])`	Массив из единиц заданной `формы` и `типа`
`identity(n)`	Возвращает двумерный массив формы `(n, n)`

Модуль `Numeric` также определяет константы `e` (число e) и `pi` (число пи).

Конечно, научиться применять эти функции можно только в процессе практики, при решении конкретной задачи.

11.2. Модуль *LinearAlgebra*

Модуль `LinearAlgebra` предоставляет доступ к некоторым часто использующимся в линейной алгебре алгоритмам. Предполагается, что читатель хоть немного знаком с линейной и матричной алгеброй и без труда найдет применение описываемым ниже функциям.

Функция `determinant()` находит *детерминант (определитель) матрицы*:

```
>>> import Numeric, LinearAlgebra
>>> print LinearAlgebra.determinant(
...     Numeric.array([[1, -1],
...                    [2, 3]]))
5
```

Функция `solve_linear_equations()` позволяет решать линейные уравнения вида `ax=b` по данным ей аргументам `a` и `b`. Например:

```
>>> import Numeric, LinearAlgebra
>>> a = Numeric.array([[1.0, 2.0],
...                    [0.0, 1.0]])
>>> b = Numeric.array([1.1, 0.5])
>>> x = LinearAlgebra.solve_linear_equations(a, b)
>>> print "x =", x
x = [ 0.1  0.5]
>>> print "Проверка:", Numeric.dot(a, x) - b
Проверка: [ 0.  0.]
```

В случае, когда матрица `a` имеет нулевой определитель, возбуждается исключение `LinearAlgebraError` с уточнением: `Singular matrix`.

Функция `inverse()` позволяет обращать матрицу. Только не следует решать линейные уравнения с помощью `LinearAlgebra` умножением на обратную матрицу, ведь она сама определена так:

```
def inverse(a):
    return solve_linear_equations(a, Numeric.identity(a.shape[0]))
```

Функция `eigenvalues()` находит собственные значения матрицы, а `eigenvectors()` — пару: собственные значения, собственные векторы. Следующий пример демонстрирует эти две функции:

```
from Numeric import *
from LinearAlgebra import *
a = array([[-5, 2],
           [2, -2]])
lmd = eigenvalues(a)
print "Собственные значения:", lmd
(lmd, v) = eigenvectors(a)
print "Собственные вектора:"
print v
print "Проверка:", dot(a, v[0]) - v[0] * lmd[0]
```

Результат выполнения:

```
Собственные значения: [-1. -6.]
Собственные вектора:
[[ 0.4472136   0.89442719]
 [-0.89442719  0.4472136 ]]
Проверка: [  6.66133815e-16  -3.33066907e-16]
```

В последней строке числа совсем маленькие (практически нули), значит, собственные числа и вектора найдены верно.

Еще одна функция, использующаяся для исследования матриц — сингулярное разложение матрицы.

```
singular_value_decomposition(A[, full]) -> (U, d, Vt)
```

где `A` — исходная матрица, которую представляют в виде `U*diag(d)*Vt`; `full` — влияет на размерности результирующих матриц, если `A` не является квадратной, по умолчанию 0; `U` — левый множитель (ортогональная матрица); `d` — вектор диагонали; `Vt` — правый множитель (ортогональная матрица).

Рассмотрим пример:

Листинг 11.2. Сингулярное разложение

```
from Numeric import *
from LinearAlgebra import *
```

```
def diag(d):
  """Породить диагональную матрицу по заданной диагонали"""
  n = d.shape[0]
  r = zeros([n]*2, d.typecode())
  ravel(r)[0::n+1] = d
  return r

A = array([[1,  2],
           [-1, 4]])
(U, d, Vt) = singular_value_decomposition(A)
print "Проверяем ортогональность матриц U и Vt:"
print abs(determinant(U)), abs(determinant(Vt))
print "Проверяем правильность сингулярного разложения:"
print dot(dot(U, diag(d)), Vt) - A
```

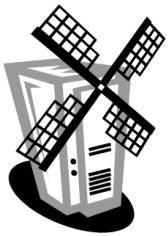
Результат выполнения:

```
Проверяем ортогональность матриц U и Vt:
1.0 1.0
Проверяем правильность сингулярного разложения:
[[ 0.  0.]
 [ 0.  0.]]
```

Замечание

Модуль `LinearAlgebra` является облегченной версией (эмуляцией) модуля `LinAlg`. Последний же является высокоуровневым интерфейсом к хорошо отлаженной и эффективной библиотеке `LAPACK`, написанной на FORTRAN или C.

Глава 12



Обработка изображений: Python Image Library

Библиотека Python Image Library (PIL), доступная в Internet по адресу **http://www.pythonware.com/products/pil/**, добавляет в Python возможности программной обработки растровых графических изображений. Речь идет о следущих возможностях:

- чтение и запись файлов в различных графических форматах;
- эффективное хранение изображения в различных цветовых системах;
- редактирование изображения с помощью внесения в него примитивных графических элементов (линий, прямоугольников, овалов и т. п.) посредством класса `ImageDraw`;
- применение фильтров для изменения изображения (класс `ImageFilter`);
- операции по комбинированию двух графических изображений (класс `ImageChops`);
- поддержка классов `PhotoImage` и `BitmapImage` для модуля `Tkinter`;
- поддержка шрифтов (в собственном формате или формате TrueType).

Ниже мы коснемся лишь некоторых из этих возможностей. При описании PIL автор сверялся с исходными кодами библиотеки, а также с оригинальной документацией разработчиков Python Image Library, Фредрика Лунда (Fredrik Lundh) и Secret Labs AB. Документация может быть найдена в Internet по указанному выше адресу.

12.1. Модуль *Image*

Модуль `Image` является краеугольным камнем для всей библиотеки. Он определяет одноименный класс, объекты которого представляют графические изображения. Получить графическое изображение можно, загрузив его из файла, получив в результате некоторой операции над изображениями или нарисовав его с чистого листа.

12.1.1. Примеры

В следующем примере мы загружаем изображение из файла lena.ppm в формате PPM (Portable Pix Map, переносимое отображение пикселов), выводим различные данные о нем и записываем в другой файл в формате PNG (Portable Network Graphics, переносимая сетевая графика). Это изображение хорошо известно всем исследователям алгоритмов машинной графики, и его можно найти в поставке PIL в каталоге `Images`. В примере подразумевается, что изображение было скопировано в текущий рабочий каталог.

Листинг 12.1. Файл lena1.py. Получение характеристик изображения

```
import Image
def descr(img):
  """Характеристики изображения"""
  print "Источник: %s" % img.filename
  print "Формат: %s (%s)" % (img.format, img.format_description)
  print "Размер: %sx%s" % img.size
  print "Режим: %s" % img.mode
img = Image.open("lena.ppm")
descr(img)
img.save("lena.png")
pngimg = Image.open("lena.png")
descr(pngimg)
```

Результат выполнения:

```
Источник: lena.ppm
Формат: PPM (Pbmplus image)
Размер: 128x128
Режим: RGB
Источник: lena.png
Формат: PNG (Portable network graphics)
Размер: 128x128
Режим: RGB
```

Режим `RGB` соответствует изображению в TrueColor.

Следующий метод будет очень полезен для Web-мастеров, т. к. автоматически создает уменьшенную копию изображения ("ноготок"):

Листинг 12.2. Файл lena2.py. Создание "ноготка" изображения

```
import Image
img = Image.open("lena.ppm")
img.thumbnail((48, 48))
img.save("lenochka.jpg", "JPEG")
```

А в следующем примере в листинге 12.3 изображение сначала увеличивается в три раза, затем поворачивается на 30 градусов против часовой стрелки и записывается в файл в формате JPEG. При увеличении изображение сглаживается методом `BICUBIC`.

Листинг 12.3. Файл lena3.py. Увеличение и поворот изображения

```
import Image
img = Image.open("lena.ppm")
img.resize((128*4, 128*4), Image.BICUBIC).rotate(30).save("elena.jpg")
```

12.1.2. Функции

В модуле `Image` определены некоторые функции, которые мы рассмотрим ниже.

- `Image.new(mode, (w, h)[, color]) -> image`

  Создает новое изображение в режиме `mode` шириной `w`, высотой `h`. Необязательный параметр `color` задает кортеж (или одиночное значение) с цветом изображения.

- `Image.open(infile) -> image`

  Открывает файл с именем infile, содержащий графическое изображение. В качестве infile можно задать файлоподобный объект, имеющий методы `read()`, `tell()` и `seek()`.

- `Image.blend(image1, image2, a) -> image`

  Получает новое изображение путем смешения двух других изображений. Число с плавающей точкой `a` указывает пропорции смешения. При `a`, равном нулю, `image` идентичен `image1`, при `a`, равном единице, — `image2`. Промежуточные значения дают смесь двух изображений. Разумеется, изображения должны иметь одинаковый размер.

- `Image.composite(image1, image2, mask) -> image`

  Аналогична `Image.blend()`, но в качестве пропорции используется изображение `mask`. Режим `mask` может быть `"1"`, `"L"` или `"RGBA"`.

- `Image.eval(func, image) -> image1`

  Отображает значения пикселов с помощью функции `func()`. Функция `func()` одна для всех каналов и вызывается только для всех значений пикселов, а не для каждого пиксела отдельно.

- `Image.fromstring(mode, (w, h), data) -> image`

  Формирует изображение размером `w` на `h` на основе данных из строки `data`.

- `Image.merge(mode, (ch1, ch2, ...)) -> image`

  Формирует изображение из отдельных каналов. Число заданных в кортеже изображений должно соответствовать режиму `mode`. Изображения отдельных каналов `ch1`, `ch2` и т. д. должны быть одинакового размера. Этот размер и будет иметь результирующее изображение `image`.

12.1.3. Методы объектов класса *Image*

В табл. 12.1 описаны многие методы объектов класса `Image`.

Таблица 12.1. *Методы объектов класса* `Image`

Метод	Описание
`convert(mode [, matrix]) -> image`	Преобразует изображение в другой режим хранения пикселов. Параметр `mode` может быть одним из обозначенных в табл. 12.2. Параметр `matrix` — кортеж из четырех или двенадцати элементов, задающий матрицу преобразования цветовых координат (используется для преобразования в `"L"` или `"RGB"`)
`copy() -> image`	Копирует изображение
`crop((left, top, right, bottom)) -> image`	Кадрирует изображение по заданным в параметре-кортеже координатам левого верхнего и правого нижнего углов
`draft(mode, size)`	Конфигурирует загрузчик файла на определенные значения режима `mode` и размера. Имеет действие только до применения метода `load()` или других методов, которые реально загружают изображение в память
`filter(F) -> image`	Возвращает изображение, являющееся результатом применения фильтра `F` (из модуля `ImageFilter`, см. табл. 12.3)
`fromstring (dat[a, decoder, parameters])`	Аналог функции `Image.fromstring()`, но загружает данные в сам объект

Таблица 12.1 *(продолжение)*

Метод	Описание
`getbands() -> tuple`	Возвращает кортеж с названиями каналов изображения. Например, для изображения в режиме `RGBA` метод даст: `('R', 'G', 'B', 'A')`
`getbbox() -> (left, top, right, bottom)`	Возвращает координаты ограничивающего прямоугольника изображения (то есть ненулевой области изображения)
`getdata() -> ic`	Возвращает содержимое изображения. К результату можно обратиться по индексу пиксела, получив кортеж значений различных каналов изображения. Весь массив пикселов выпрямлен в одномерный массив построчно
`getextrema() -> (min, max)`	Для изображений с одним каналом возвращает значения минимума и максимума
`getpixel((x, y)) -> color`	Возвращает кортеж со значениями для всех каналов изображения для пиксела с координатами `x`, `y` или одно значение, если изображение имеет всего один канал
`histogram([mask]) -> counts`	Возвращает гистограмму для данного изображения в виде списка с количеством пикселов, имеющих определенное значение от нуля до 255. Для многоканальных изображений гистограммы для всех каналов конкатенируются в один список. Например, для изображения в режиме `RGB` длина списка `counts` будет равна 768 (трижды по 256). Если задан параметр `mask`, он должен быть черно-белым изображением и указывать те пикселы, которые необходимо подсчитать для гистограммы
`load()`	Загружает данные изображения в память (обычно вызывается автоматически при первой необходимости)
`paste(image, box[, mask])`	Вставляет изображение `image` в данное изображение по месту, заданному `box`. Параметр `box` может быть кортежем из двух значений (координаты левого верхнего угла) или четырех значений (координаты верхнего левого и правого нижнего углов). Изображение `image` должно соответствовать `box` по размерам. При вставке автоматически выполняется приведение изображения к другому режиму хранения пикселов. Если задан параметр `mask`, пикселы изменяются согласно маске
`paste(color, box[, mask])`	То же, но прямоугольная область, заданная `box`, заливается цветом `color`. Цвет задается кортежем (для многоканальных изображений) или отдельным значением

***Таблица 12.1** (продолжение)*

Метод	Описание
`point(table) -> image, point(func) -> image`	Задает преобразование значений отдельных пикселов с помощью таблицы `table` или функции одного аргумента `func`. Таблица должна содержать значения для всех каналов. Функция отображает старое значение пиксела в новое
`putalpha(img)`	Присваивает альфа-каналу данного изображения значение из одноканального изображения `img`
`putdata(data[, scale [, offset]])`	Наполняет изображение данными из последовательности `data` (начиная с левого верхнего угла, построчно). Параметры `scale` и `offset` по умолчанию равны `1.0` и `0.0` соответственно и служат для преобразования значений `data` (исходное значение умножается на `scale` и к нему прибавляется `offset`)
`putpalette(seq)`	Присваивает изображениям, хранимым в режимах `"L"` или `"P"`, палитру `seq` (последовательность целых чисел или строка байтов). Палитра должна иметь 256 значений на каждый канал
`putpixel((x, y), color)`	Присваивает пикселу с координатами `(x, y)` цвет `color` (кортеж со значениями всех каналов)
`resize((w, h) [, F])`	Изменяет размер изображения, возвращая копию изображения шириной `w` и высотой `h` пикселов. В качестве сглаживающего фильтра `F` можно указать: `NEAREST` (ближайшая точка, используется по умолчанию), `BILINEAR` (линейная интерполяция по четырем соседним точкам) или `BICUBIC` (кубическая интерполяция по 16 соседним точкам)
`save(outfile [[, format], опции])`	Сохраняет изображение в файле с именем outfile (или файлоподобном объекте, открытом в режиме `"wb"` и имеющем методы `write()`, `seek()`, `tell()`). Параметр `format` может указать формат файла (если он не задан, PIL пытается выяснить формат на основании названия файла). Именованные параметры `опции` задают опции для конкретного формата. Например, при сохранении в формате PNG можно указать опцию `optimize`
`seek(n) -> image`	Устанавливает указатель на `n`-й кадр изображения. Например, `n`-й кадр анимированного GIF. Нумерация начинается с нуля
`show()`	Показывает изображение на экране с помощью внешней программы (`xv` на UNIX, Paint или другая программа для показа файлов формата bmp на Windows). Обычно используется для отладки
`split() -> (ch1, ch2, ...)`	Возвращает кортеж изображений, соответствующий каналам исходного изображения

Таблица 12.1 *(окончание)*

Метод	Описание
`tell() -> n`	Возвращает текущий номер кадра. После загрузки изображения с несколькими кадрами указатель кадров устанавливается на ноль
`thumbnail((w, h))`	Создает на месте данного изображения пропорционально уменьшенное изображение, шириной не более `w` и высотой не более `h`
`tostring() -> s`	Возвращает изображение в виде строки байтов
`transform((w, h), way, data[, F]) -> image`	Создает преобразованное изображение размера `(w, h)` в том же режиме, что и данное. В качестве способа преобразования `way` можно указать `EXTENT` (вырезка прямоугольной области), `AFFINE` (аффинное преобразование), `QUAD` (преобразует четырехугольник в прямоугольник), `MESH` (преобразует несколько четырехугольников за одну операцию). Параметр `F` имеет то же значение, что и у `resize()`. Для метода `EXTENT` параметр `data` должен содержать координаты прямоугольной области `(x0, y0, x1, y1)`, которая при преобразовании перейдет в `(0, 0, w, h)` в результирующем изображении. Для `AFFINE` данные должны быть кортежем из шести элементов `(a1, b1, c1, a2, b2, c2)` для матрицы аффинного преобразования. Каждый пиксел с координатами `(x, y)` в результирующем изображении строится на основе ближайших к точке `(a1*x+b1*y+c1, a2*x+b2*y+c2)` пикселов. Дата для метода `QUAD` — восьмиэлементный кортеж `(x0, y0, x1, y1, x2, y2, x3, y3)` с координатами вершин четырехугольника в исходном изображении. Для `MESH` данные представляют собой список с координатами результирующих прямоугольников и исходных четырехугольников
`transpose(way) -> image`	Возвращает изображение, полученное путем поворота или отражения исходного изображения. В качестве `way` можно указать `ROTATE_90` (поворот на 90° против часовой стрелки), `ROTATE_180`, `ROTATE_270`, `FLIP_LEFT_RIGHT` (отражение относительно вертикальной оси), `FLIP_TOP_BOTTOM` (отражение относительно горизонтальной оси)
`verify()`	Пытается проверить целостность изображения в файле. Возбуждает исключение, если в формате найдены ошибки

В табл. 12.2 приведены возможные значения для параметра `mode`, используемого некоторыми методами экземпляров класса `Image`.

Таблица 12.2. *Доступные PIL режимы хранения изображения*

Обозначение	Число битов на пиксел	Описание
"1"	1	Черно-белое изображение, хранится как и "L"
"L"	8	Изображение в градациях серого
"P"	8	Отображается на любой другой режим с помощью цветовой палитры
"RGB"	24	TrueColor
"RGBA"	32	TrueColor с маской прозрачности
"CMYK"	32	Цветоделение CMYK
"YCbCr"	24	Формат цветного видео
"I"	32	Пикселы в целых числах
"F"	32	Пикселы в вещественных числах

12.2. О других модулях

В качестве примера посмотрим, как PIL интегрируется с применяемым в Python стандартным графическим интерфейсом Tkinter. Пример в листинге 12.4 демонстрирует применение различных фильтров к одному и тому же изображению из файла lena.ppm, которое мы уже неоднократно использовали ранее.

Листинг 12.4. Файл pildemo.py. Демонстрация интеграции Tkinter и PIL

```
import Tkinter, Image, ImageTk, ImageFilter

def show_name(event):
  """Показ соответствующей подписи"""
  n = (event.y/128)*3 + event.x/128
  status.config(text=example[n])

# инициализация Tkibter и создание рисунка и подписи
tk = Tkinter.Tk()
c = Tkinter.Canvas(tk, width=128*3, height=128*3)
c.pack()
status = Tkinter.Label(tk)
status.pack()
c.bind("<Motion>", show_name)
```

```
src_img = Image.open("lena.ppm")

# названия фильтров, использованных в примере
example = ["SOURCE",     "BLUR",              "CONTOUR",
           "DETAIL",     "EDGE_ENHANCE_MORE", "EMBOSS",
           "FIND_EDGES", "SHARPEN",           "SMOOTH_MORE"]

img = {}  # словарь для хранения изображений
# располагаем обработанные изображения на рисунке
for i in range(3):
  for j in range(3):
    n = i*3 + j
    if example[n] is not "SOURCE":
      # берем фильтр с соответствующим названием
      F = ImageFilter.__dict__[example[n]]
      img[n] = ImageTk.PhotoImage(src_img.filter(F))
    else:
      img[n] = ImageTk.PhotoImage(src_img)
    # размещаем изображение на рисунке
    c.create_image(j*128, i*128, image=img[n], anchor='nw')

tk.mainloop()
```

На рис. 12.1 приведен результат работы программы `pildemo.py` (курсор находился на среднем изображении в правой колонке, поэтому подпись показывает `"EMBOSS"`).

В табл. 12.3 приведены фильтры, определенные в модуле `ImageFilter`.

Таблица 12.3. *Фильтры модуля* `ImageFilter`

Фильтр	Описание
`BLUR`	Смазывание
`CONTOUR`	Выделение контуров
`DETAIL`	Усиление деталей изображения
`EDGE_ENHANCE`	Усиление краев
`EDGE_ENHANCE_MORE`	Большое усиление краев
`EMBOSS`	Получить выпуклое изображение
`FIND_EDGES`	Поиск краев

Таблица 12.3 *(окончание)*

Фильтр	Описание
SHARPEN	Повышение резкости
SMOOTH	Сглаживание
SMOOTH_MORE	Сильное сглаживание

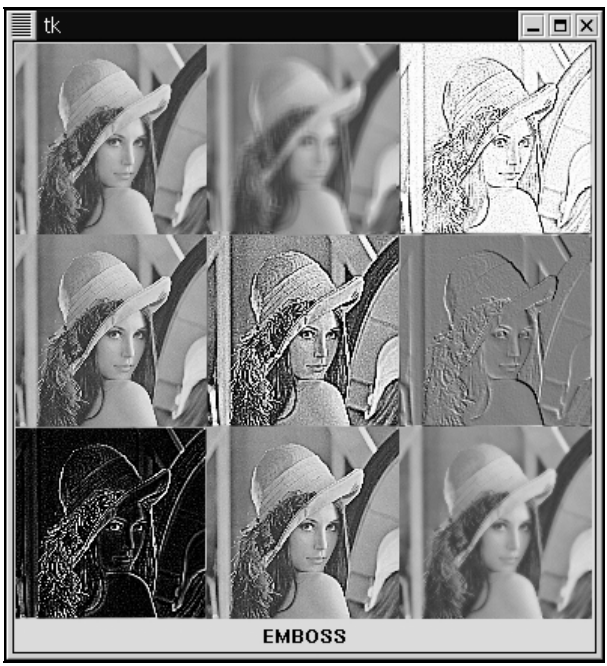


Рис. 12.1. Результат работы программы `pildemo.py`

Следующий пример показывает, что изображение можно задавать в тексте программы, если использовать формат XPM (X11 Pixel Map). Такое изображение можно править в обычном текстовом редакторе!

Листинг 12.5. Файл pilbutton.py. Хранение изображения в программе

```
import Tkinter, Image, ImageTk, StringIO
tk = Tkinter.Tk()
# Читаем изображение из файла-строки:
src_img = Image.open(
  StringIO.StringIO("""/* XPM */
```

```
static char * doc1_xpm[] = {
"19 23 3 1",
"       c None",
".      c #000000",
"+      c #FFFFFF",
"+++++++++++++++++++",
"+++++++++++++++++++",
".............++++++",
".++++++++++..+++++",
".++++++++++.+.++++",
".+++.+.+.+.+.++.+++",
".++........+.+++.++",
".++++++++++......+",
".++.........+++....",
".++++++++++++++..",
".++............++..",
".++++++++++++++..",
".++...........+++..",
".++++++++++++++..",
".++..........++++..",
".++++++++++++++..",
".++...........+++..",
".++++++++++++++..",
".++++++++++++++..",
"...................",
"+..................",
"+++++++++++++++++++",
"+++++++++++++++++++"};
""")
img = ImageTk.PhotoImage(src_img)
c = Tkinter.Button(tk, image=img)
c.pack()
tk.mainloop()
```

Глава 13



Связь с базами данных

Для языка Python разработан специальный программный интерфейс с базами данных *DB API*. На момент написания книги DB API имел версию 2. Все модули, которые служат для сопряжения Python-программ с базами данных, должны придерживаться этого API. Наличие единого API позволяет легче переходить с одной системы управления базами данных (*СУБД*) на другую.

В этой главе мы рассмотрим как сам программный интерфейс к базам данных, так и пару модулей для связи с базами данных MySQL и PostgreSQL.

13.1. Описание DB API 2.0

DB API 2.0 регламентирует интерфейсы модуля для сопряжения с базой данных, объекта для соединения с базой, объекта-курсора для указания текущей обрабатываемой записи, объектов, различных для типов данных и их конструкторов, а также содержит некоторые предложения по реализации модулей. С современным состоянием поддержки баз данных в Python можно познакомиться по адресу **http://www.python.org/topics/database/**.

Доступ к базе данных осуществляется с помощью объектов-соединений. Модуль должен описывать функцию-конструктор.

```
connect(параметры) -> connobj
```

Функция для создания объектов-соединений. `Параметры` обычно зависят от используемой базы данных. Рекомендуемые названия именованных аргументов и их порядок следующий: `dsn` — название источника данных в виде строки; `user` — имя пользователя; `password` — строка с паролем; `host` — адрес хоста, на котором работает СУБД; `database` — имя базы данных.

В табл. 13.1 представлены константы, которые должны быть определены в модуле, поддерживающем DB API 2.0.

Таблица 13.1. *Константы DB API 2.0*

Константа	Описание
`Apilevel`	Строковая константа, указывающая на используемую модулем версию API. (На момент написания книги это могли быть строки `"1.0"` или `"2.0"`.)

Таблица 13.1 *(окончание)*

Константа	Описание
`threadsafety`	Целая константа, описывающая возможности модуля при использовании потоков управления: `0` — модуль не поддерживает потоки; `1` — потоки могут разделять модуль, но не соединения; `2` — потоки могут разделять модуль и соединения; `3` — потоки могут разделять модуль, соединения и курсоры. Под разделением здесь подразумевается, что потоки могут использовать упомянутые ресурсы без применения семафоров. Работу с внешними ресурсами не всегда можно сделать совместимой с потоками, т. к. они могут зависеть от внешних факторов
`paramstyle`	Строковая константа, обозначающая тип используемых пометок при подстановке параметров. Значение `"format"` соответствует использованию форматирования, предусмотренного стандартом ANSI для языки C (например, `"%s"`), `"pyformat"` позволяет задавать именованные спецификаторы формата, как в Python (`"%(item)s"`), `"qmark"` использует знаки `"?"` для пометки мест подстановки параметров, `"numeric"` означает использование номеров позиций (`":1"`), `"named"` позволяет задать имена подставляемых параметров (`":name"`)

Модуль должен определять ряд исключений для обозначения исключительных ситуаций: `Warning` (предупреждение), `Error` (ошибка), `InterfaceError` (ошибка интерфейса к БД), `DatabaseError` (ошибка, относящаяся к базе данных. Все перечисленные далее исключения являются подклассами этого исключения), `DataError` (ошибка обработки данных), `OperationalError` (ошибка в работе или соединении с базой данных), `IntegrityError` (ошибка целостности базы данных), `InternalError` (внутренняя ошибка базы данных), `ProgrammingError` (программная ошибка, например, ошибка в синтаксисе SQL-запроса), `NotSupportedError` (исключение, возбуждаемое при отсутствии поддержки затребованного свойства).

В табл. 13.2 представлены методы объектов-соединений. Как уже говорилось выше, эти объекты появляются в результате выполнения функции `connect()`.

Таблица 13.2. *Методы объектов-соединений*

Метод	Описание
`close()`	Закрывает соединение с базой данных. После этого с помощью данного объекта-соединения будет невозможно обращаться к базе данных
`commit()`	Завершает транзакцию

Таблица 13.2 *(окончание)*

Метод	Описание
`rollback()`	Откатывает начатую транзакцию (восстанавливает исходное состояние). Закрытие соединения при незавершенной транзакции автоматически производит откатку транзакции
`cursor()`	Возвращает объект-курсор, использующий данное соединение. Если база данных не поддерживает курсоры, модуль сопряжения должен их имитировать другими средствами, чтобы соответствовать DB API

Под *транзакцией* здесь понимается группа из одной или нескольких операций, которые изменяют базу данных. Причем, одна транзакция соответствует логически неделимой операции. Например, при переводе денег с одного счета на другой операции по уменьшению первого счета и увеличению второго являются транзакцией. Понятно, что транзакцию нельзя выполнить наполовину: это было бы некорректно. Методы `commit()` и `rollback()` позволяют задавать границы транзакции в явном виде. Не все базы данных поддерживают механизм транзакций, поэтому указанные методы не всегда имеют смысл.

Атрибуты объектов-курсоров кратко описаны в табл. 13.3.

Таблица 13.3. *Некоторые атрибуты объектов-курсоров*

Атрибут	Описание
`arraysize`	Атрибут, указывающий количество записей, возвращаемых методом `fetchmany()`. По умолчанию равен `1`
`callproc(proc [, params])`	Вызывает хранимую процедуру `proc` с параметрами из изменчивой последовательности `params`. Хранимая процедура может изменить значения некоторых параметров, оставив другие нетронутыми. Кроме того, может быть возвращен результат, доступный с помощью `fetch`-методов
`close()`	Закрывает объект курсор
`execute(op [, params])`	Исполняет запрос к базе данных или команду СУБД. Параметры `params` могут быть представлены в специфичной для базы данных нотации в соответствии с атрибутом `paramstyle`, описанным выше
`executemany (op, seq_of_ params)`	Выполняет серию запросов или команд, используя один и тот же шаблон с разными параметрами. Параметр `seq_of_params` задает последовательность наборов параметров
`fetchall()`	Возвращает все (или все оставшиеся) записи результата запроса

Таблица 13.3 *(окончание)*

Атрибут	Описание
`Fetchmany ([size])`	Возвращает следующие несколько записей из результатов запроса в виде последовательности последовательностей. Пустая последовательность означает отсутствие данных. Необязательный параметр `size` указывает количество возвращаемых записей (записей может быть возвращено и меньше этого количества). По умолчанию он равен атрибуту `arraysize` объекта-курсора
`fetchone()`	Возвращает следующую запись (в виде последовательности) из результата запроса или `None`, если данных больше нет
`nextset()`	Переводит курсор к следующему набору данных, полученному в результате запроса (возможно, оставляя часть записей в предыдущем наборе). Если больше наборов нет, возвращается `None`. Не все базы данных поддерживают несколько наборов результатов
`rowcount`	Количество записей, полученных или затронутых в результате выполнения последнего запроса. Равен `-1` в случае отсутствия `execute`-запросов или невозможности указать количество записей
`setinputsizes (sizes)`	Предопределяет области памяти для параметров, используемых в операциях. Аргумент `sizes` задает последовательность, в которой каждый элемент соответствует одному входному параметру. Элемент может являться объектом-типом соответствующего параметра или целым числом, задающим длину строки. Он также может иметь значение `None`, если о размере входного параметра ничего неизвестно или он очень большой. Метод должен быть вызван прежде, чем `execute`-методы
`setoutputsize (size[, n])`	Устанавливает размер буфера для выходного параметра с номером `n` или всех больших выходных параметров. Может использоваться, например, для больших бинарных объектов (Binary Large Object, BLOB)

В табл. 13.4 приведены предусмотренные DB API 2.0 объекты-типы, используемые для описания полей базы данных.

Таблица 13.4. *Объекты-типы для описания полей*

Объект	Что обозначает
`STRING`	Строка и символ
`BINARY`	Бинарный объект
`NUMBER`	Числа

Таблица 13.4 *(окончание)*

Объект	Что обозначает
`DATETIME`	Дата и время
`ROWID`	Идентификатор записи (ID записи)

Для обозначения отсутствующего значения в базе данных (то есть NULL-значения) используется `None`.

Кроме того, в табл. 13.5 приведены конструкторы для создания объектов некоторого типа. Модуль, поддерживающий DB API, должен их определять. Чаще всего, для функций работы со временем употребляется модуль `mx.DateTime` из состава набора модулей `MxTools`, доступного с сайта **http://www.lemburg.com/files/python/**. Он есть и на компакт-диске, прилагаемом к этой книге.

Таблица 13.5. *Конструкторы для создания объектов различных типов*

Конструктор	Какого типа объект создает
`Date(год, месяц, день)`	Дата
`Time(час, минута, секунда)`	Время
`Timestamp(год, месяц, день, час, минута, секунда)`	Дата-время
`DateFromTicks(secs)`	Дата, на основании числа секунд `secs` от начала эпохи *(см. разд. 9.5.1)*
`TimeFromTicks(secs)`	Время, то же
`TimestampFromTicks(secs)`	Дата-время, то же
`Binary(s)`	Большой бинарный объект на основании строки `s`

В следующих разделах мы рассмотрим модули для работы с базами данных MySQL и PostgreSQL. Обе эти реляционные СУБД в меньшей (MySQL) или большей (PostgreSQL) степени поддерживают стандарт SQL92 и имеют специфичные области применения. MySQL часто применяется как хранилище данных для построения динамических Web-сайтов. К этому располагают ее высокое быстродействие на операциях чтения, удобный тип данных для хранения больших бинарных объектов. PostgreSQL же является не такой быстродействующей СУБД, но зато поддерживает практически весь стандарт SQL, является объектно-ориентированной базой данных, имеет хорошие статистические возможности. Эта СУБД может применяться там, где от базы данных требуется бóльшая гибкость и весь джентльменский набор средств, свойственных коммерческим СУБД.

Разумеется, Python имеет модули для работы и с другими СУБД: Oracle, Interbase, Sybase, MS SQL Server, DB/2, Informix, Ingres, а также модули для ODBC и JDBC. Список модулей для различных баз данных можно найти в Internet по адресу **http://www.python.org/topics/database/modules.html**. Кроме того, в Python доступна и небольшая реляционная СУБД, поддерживающая многие возможности SQL и написанная целиком на Python: это набор модулей Gadfly. Удобство Gadfly в том, что эта СУБД может удовлетворить потребности программ на Python, для которых внешняя СУБД является излишней роскошью.

13.2. Модуль *MySQLdb*

СУБД MySQL имеется на прилагаемом компакт-диске. Для связи с СУБД MySQL разработан модуль `MySQLdb`. Ниже мы предполагаем, что MySQL-сервер уже установлен и может быть настроен таким образом, что пользователь `rnd` с паролем `"secretpass"` имеет полный доступ к базе `test` (эта база доступна сразу после установки MySQL-сервера). Также предполагается, что модуль `MySQLdb` тоже установлен в системе. Для настройки требуемого доступа суперпользователь должен выполнить специальную команду (ниже приводится программа на Python, хотя для работы с MySQL можно использовать специальную оболочку для командной строки):

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(db="mysql", user="root")
>>> cur = conn.cursor()
>>> cur.execute("""GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP
...    on test.* to rnd@localhost identified by 'secretpass';""")
0L
>>> cur.close()
>>> conn.close()
```

Буквально приведенные команды означают следующее: пользователь `root` подключается к базе с названием `mysql`, заводит объект-курсор `conn` и выполняет команду. Эта команда разрешает операции (команды SQL) `SELECT` (выбрать), `INSERT` (добавить), `UPDATE` (обновить), `DELETE` (удалить), `CREATE` (создать) и `DROP` (выбросить) в базе `test` пользователю `rnd` с локальной машины, если он правильно укажет пароль `"secretpass"`. После чего курсор и соединение с базой данных закрываются, а пользователь `rnd` может приступать к работе.

Для работы с современными реляционными СУБД обычно необходимо знать язык SQL (Structured Query Language, язык структурированных запросов). Приведенные ниже примеры будут снабжены обширными комментариями, однако для практических применений модуля `MySQLdb` лучше все-таки изучить SQL.

В качестве примера мы рассмотрим следующую задачу. Программа будет читать все ру-файлы в указанном каталоге (в каталоге со стандартными модулями Python) и заполнять базу данных по определяемым в ру-файлах функциям. Такой необычный пример позволит не только не искать внешние, достаточно искусственные источники данных, но и поможет лучше понять структуру модулей.

В начале нужно создать в базе данных все необходимые таблицы (листинг 13.1).

Листинг 13.1. Файл db1.py. Работа с базой данных: создание таблиц

```
import MySQLdb

conn = MySQLdb.connect(db="test", user="rnd", passwd="secretpass")
cur = conn.cursor()
try:
  cur.execute(
  """CREATE TABLE modules (           -- создаем таблицу modules
     name VARCHAR(100),               -- имя модуля
     last_modification TIMESTAMP,     -- время последней модификации
     size INT,                        -- размер в байтах
     lines_of_code INT,               -- количество строк кода
     path TEXT                        -- путь к модулю
   )""")
except MySQLdb.OperationalError:
  print "Не удалось создать таблицу modules"
cur.close()
conn.close()
```

Теперь мы можем наполнить наши таблицы конкретным содержимым (листинг 13.2).

Листинг 13.2. Файл db2.py. Работа с базой данных: наполнение данными

```
import MySQLdb, glob, os, stat, sys, mx.DateTime

conn = MySQLdb.connect(db="test", user="rnd", passwd="secretpass")
cur = conn.cursor()
# удаляем все записи для чистоты эксперимента:
cur.execute("""DELETE FROM modules;""")
```

```
target_dir = sys.path[1]  # один из каталогов для поиска модулей

file_names = glob.glob(os.path.join(target_dir, "*.py"))
for file_name in file_names:
  file_stat = os.stat(file_name)
  file_contents = open(file_name, "rt").readlines()
  cur.execute(  # вставляем запись об очередном файле
  """INSERT INTO modules VALUES (%s, %s, %s, %s, %s);""",
  (os.path.basename(file_name)[:-3],         # имя модуля
   mx.DateTime.gmtime(file_stat[stat.ST_MTIME]), # время последней
                                             # модификации
   file_stat[stat.ST_SIZE],                  # размер в байтах
   len(file_contents),                       # количество строк кода
   file_name,                                # путь к модулю
   )
  )
cur.close(); conn.close()
```

Настало время сделать выборки из нашей базы данных. В листинге 13.3 представлено и прокомментировано несколько примеров таких выборок.

Листинг 13.3. Файл db3.py. Работа с базой данных: выборки

```
import MySQLdb

conn = MySQLdb.connect(db="test", user="rnd", passwd="secretpass")
cur = conn.cursor()

cur.execute("""SELECT COUNT(name) FROM modules;""")
print "Количество модулей:", cur.fetchall()[0][0]

cur.execute(
"""SELECT COUNT(*) FROM modules WHERE lines_of_code > 1000;""")
print "Количество очень больших модулей:", cur.fetchall()[0][0]

cur.execute(
"""SELECT name, lines_of_code, size FROM modules
    WHERE lines_of_code > 1000;""")
print "Список очень больших модулей:"
```

```
for record in cur.fetchall():
  print "%-10s : %10s : %10s" % record

cur.close(); conn.close()
```

Результат выполнения:

```
Количество модулей: 160
Количество очень больших модулей: 6
Список очень больших модулей:
doctest    :       1118 :      36505
imaplib    :       1125 :      34010
mhlib      :       1003 :      33258
pydoc      :       1878 :      75893
urllib     :       1383 :      47589
urllib2    :       1138 :      37454
```

При работе с базой данных бывает необходимо корректировать их содержимое. В листинге 13.4 показано обновление и удаление записей из базы данных.

Листинг 13.4. Файл db4.py. Работа с базой данных: обновление

```
import MySQLdb, time

conn = MySQLdb.connect(db="test", user="rnd", passwd="secretpass")
cur = conn.cursor()

# обновляем все значения поля last_modification в таблице modules
cur.execute(
  """UPDATE modules SET last_modification=CURRENT_TIMESTAMP;""")

cur.execute(
  """SELECT name FROM modules
  WHERE size < 1000 OR lines_of_code < 40;""")

print "Будет удалена информация о следующих модулях:"
for (name,) in cur.fetchall():
  print name,

# удаляем данные о модулях маленького размера
cur.execute(
```

```
  """DELETE FROM modules
  WHERE size < 1000 OR lines_of_code < 40;""")

cur.close(); conn.close()
```

Результат выполнения:

```
Будет удалена информация о следующих модулях:
TERMIOS copy_reg dbhash re statvfs tty
```

Те, кто знаком с SQL, уже догадались, что с помощью `MySQLdb` можно давать SQL-запросы к базе данных и получать ответы в виде последовательности кортежей Python. Конечно, модуль `MySQLdb` определяет еще некоторые другие полезные функции, но они не являются частью DB API 2.0 и потому их не стóит применять в программах, которые пишутся с расчетом использовать одну из нескольких СУБД. Каждая СУБД имеет свои особенности. Хотя многие из них поддерживают стандарт SQL-92, добиться максимальной производительности можно, если использовать специфические для каждой СУБД "хитрости".

Рассмотрение примера (в частности, заполнение данных о функциях) мы продолжим в следующем разделе.

13.3. Модуль для работы с PostgreSQL

СУБД PostgreSQL является еще одной свободно распространяемой реляционной СУБД. В отличие от MySQL, СУБД PostgreSQL имеет более широкую поддержку стандарта SQL, а также обладает очень многими свойствами больших коммерческих продуктов, таких как Oracle или Sybase. Для работы с PostgreSQL из Python существует несколько наборов модулей. Ниже мы рассмотрим один из них — модуль `PyGreSQL`. Этот набор состоит из модулей `pgdb` (реализует DB API 2.0) и `pg` (реализует более высокоуровневый интерфейс к базе данных, основан на `pg`). Эти модули используют модуль-расширение `_pg`, написанный на C. Описываемый модуль поставляется вместе с СУБД PostgreSQL.

В качестве первого примера рассмотрим перенос базы данных из MySQL в PostgreSQL. Следует иметь в виду, что PostgreSQL поддерживает транзакции и следует явно вызывать `commit()`, когда мы хотим, чтобы изменения в базе данных действительно произошли. Транзакции — очень важный элемент работы с базой данных, т. к. это — один из элементов обеспечения условий *целостности данных*.

Листинг 13.5. Файл pg1.py. Создание таблиц в PostgreSQL

```
import pgdb

conn = pgdb.connect(database="template1")
cur = conn.cursor()

cur.execute(
"""CREATE TABLE modules (           -- создаем таблицу modules
     name VARCHAR(100),             -- имя модуля
     last_modification TIMESTAMP,   -- время последней модификации
     size INT,                      -- размер в байтах
     lines_of_code INT,             -- количество строк кода
     path TEXT);                    -- путь к модулю
     """)
cur.close()
conn.commit()  # завершаем транзакцию
conn.close()
```

Листинг 13.6 переводит данные таблицы `modules` из MySQL в PostgreSQL. Обратите внимание, что данные фактически переносятся один в один. Даже поле `last_modification` перешло один в один благодаря тому, что модуль `mx.DateTime` является рекомендованным модулем для представления информации о времени.

Листинг 13.6. Файл pg2.py. Перенос данных из MySQL в PostgreSQL

```
import MySQLdb, pgdb

conn1 = MySQLdb.connect(db="test", user="rnd", passwd="secretpass")
cur1 = conn1.cursor()

conn2 = pgdb.connect(database="template1")
cur2 = conn2.cursor()

cur1.execute("""SELECT * FROM modules;""") # выбираем все записи
while 1:
  record = cur1.fetchone()  # читаем очередную запись
  if not record:
    break
```

```
  # готовим данные для записи в другую базу данных
  outrecord = {
     "name": record[0],
     "last_modification": record[1],
     "size": record[2],
     "lines_of_code": record[3],
     "path": record[4],
  }
  cur2.execute(  # пишем запись
  """INSERT INTO modules VALUES
     ('%(name)s', '%(last_modification)s', %(size)s,
     %(lines_of_code)s, '%(path)s');""",
  outrecord)

cur1.close(); cur2.close()
conn2.commit()
conn1.close(); conn2.close()
```

Тем не менее, такое гладкое преобразование получается далеко не всегда. СУБД зачастую имеют самые разнообразные типы для представления данных о дате и времени, больших объектов, различные ограничения на длину текста. Переводя базу данных с одной СУБД на другую, всегда проверяйте совместимость их типов данных во избежание недоразумений.

Заметьте также, что нам потребовалось преобразование из кортежа в словарь из-за разного представления спецификаций параметров (см. табл. 13.1):

```
>>> import MySQLdb, pgdb
>>> print MySQLdb.paramstyle
format
>>> print pgdb.paramstyle
pyformat
```

В листинге 13.7 мы продолжаем совершенствовать структуру базы данных, добавив новую таблицу для хранения определенных в модуле функций.

Листинг 13.7. Файл pg3.py. Добавляем таблицу с данными об определенных в модуле функциях

```
import pgdb, re
conn = pgdb.connect(database="template1")
cur = conn.cursor()
```

```
try:
  cur.execute(
  """CREATE TABLE functions (  -- создаем таблицу functions
     name VARCHAR(100),        -- имя функции
     module VARCHAR(100));     -- имя модуля, в котором она определена
     """)
except:
  print "Таблица functions уже создана."

redef = re.compile("^def\s+(\w+)\s*\(")
cur.execute("""SELECT name, path FROM modules;""")
for name, path in cur.fetchall():
  try:
    file_lines = open(path, "rt").readlines()
    for l in file_lines:
      m = redef.match(l)
      if m:
        cur.execute("""INSERT INTO functions VALUES
        ('%(name)s', '%(module)s');""",
                    {"name": m.group(1), "module": name})
    # все функции записались нормально, транзакция состоялась
    conn.commit()
  except:
    # произошла какая-то ошибка, отказываемся от транзакции
    conn.rollback()
```

В листинге 13.8 показаны примеры выборок, в которых задействованы сразу две таблицы: `modules` и `functions`.

Листинг 13.8. Файл pg4.py. Работаем с двумя таблицами

```
import pgdb
conn = pgdb.connect(database="template1")
cur = conn.cursor()
# Выбираем имена модулей и количество определенных в них функций,
# связав две таблицы по полям name и module соответственно и
# группируя по именам модулей.
# Результат сортируется по убыванию количества функций в модуле.
cur.execute(
```

```
"""SELECT modules.name, count(functions.name) FROM modules, functions
     WHERE modules.name = functions.module
     GROUP BY modules.name
     ORDER BY count(functions.name) DESC;
""")
# Ради примера, берем только первые десять записей из выборки
for mod_name, funcs in cur.fetchmany(10):
  print "%-12s: %-4i" % (mod_name, funcs)
```

Результат выполнения:

```
inspect     : 156
urllib      : 128
pydoc       : 120
string      : 104
stringold   : 104
posixpath   : 100
ntpath      : 92
dospath     : 88
macpath     : 84
calendar    : 72
```

Глава 14

Среда Zope



В этой главе мы кратко рассмотрим состав и основные возможности Zope — свободно распространяемой среды для создания Web-приложений, динамических Web-сайтов, порталов для совместной работы и многого другого.

Zope включает в себя следующие возможности:

- Web-сервер. Zope может работать как с Web-серверами, имеющими CGI, так и со своим собственным встроенным Web-сервером (ZServer);
- основанная на Web-технологиях среда разработчика. Zope позволяет создавать Web-приложения через Web-интерфейс;
- поддержка сценариев. Zope поддерживает несколько языков сценариев: Python, Perl и свой собственный *DTML (Document Template Markup Language, язык разметки шаблонов документов)*;
- база данных объектов. Zope использует в своей работе устойчивые объекты, которые хранятся в специальной базе данных. Zope имеет простой интерфейс для управления этой базой данных;
- интеграция с реляционными базами данных. Zope может хранить свои объекты и данные в обычных реляционных базах данных, таких как Oracle, Sybase, PostgreSQL, MySQL и др.

Ниже мы постараемся посмотреть на работу с Zope 2.4.1 глазами разных специалистов: администратора Zope-сервера, программиста-разработчика программных компонентов и менеджера по содержанию (content manager). От работы этих людей целиком зависит то, каким Zope предстает перед конечными пользователями — теми, для кого предназначены информационные услуги Zope-сервера.

14.1. С точки зрения администратора

У всякой многопользовательской системы должен быть человек, отвечающий за ее работоспособность, доступность определенных сервисов для определенного круга пользователей и защищенность. В обязанности администратора входит установка, переустановка Zope-сервера, заведение и удаление пользователей, назначение пользователям тех или иных прав. Ни-

же мы рассмотрим один из вариантов установки Zope-сервера, создание нового пользовательского ресурса, заведение пользователя и присвоение ему определенных прав.

14.1.1. Установка Zope

Для установки Zope нужно обзавестись дистрибутивом этого продукта на Web-сайте фирмы Zope Corporation: **http://www.zope.org**.

В UNIX процедура установки Zope с использованием собственного HTTP-сервера `ZServer` очень проста: нужно распаковать архив с бинарным дистрибутивом Zope в некоторый каталог, недоступный из WWW, и запустить сценарий `install` (в ходе установки этот сценарий даст пароль пользователя `admin` — обязательно запомните его, он потребуется для конфигурирования Zope). После этого можно исполнить сценарий `start` для старта Zope. По умолчанию Zope использует порты 8080 (для HTTP-сервера), 8021 (для FTP-сервера), а также открывает порт в коммуникационном домене UNIX для PCGI. С помощью опции `-P` можно указать другую базу для порта. Ниже будет применяться базовое значение `8100`, поэтому Zope будет использовать порты 8180 и 8121 для HTTP и FTP соответственно. В случае, если порт уже занят, Zope выдаст ошибку: Address already in use.

В Windows установка производится запуском exe-файла с установщиком. В ходе установки требуется ввести пароль администратора сервера. В будущем этот пароль необходим для входа на страницы управления сервера.

После того как ZServer запущен, можно направить Web-браузер на адрес (вместо `localhost` можно поставить адрес компьютера, на котором установлен Zope):

```
http://localhost:8180/manage
```

На рис. 14.1 показано окно браузера со страницей `http://localhost:8180/manage` — страницей для управления Zope. На эту страницу можно попасть только после ввода имени (`admin`) и пароля (его должен был сообщить вам сценарий установки Zope).

С этой страницы администратор Zope-сервера видит все находящиеся на сервере ресурсы и может манипулировать ими. Фрейм слева показывает дерево ресурсов. Если щелкнуть мышью на некотором ресурсе, его описание появляется в главном фрейме. Все ресурсы сервера растут от единого корня. Сразу после установки в корневой папке (`Root Folder`) находятся только папки `Control_Panel` (панель управления), `QuickStart` (краткое руководство) и `acl_users` (список пользователей, имеющих доступ к корневой папке). Поначалу только администратор может управлять содержанием на Zope-сервере. Он создает папки для пользователей и перепоручает им управление этими папками.

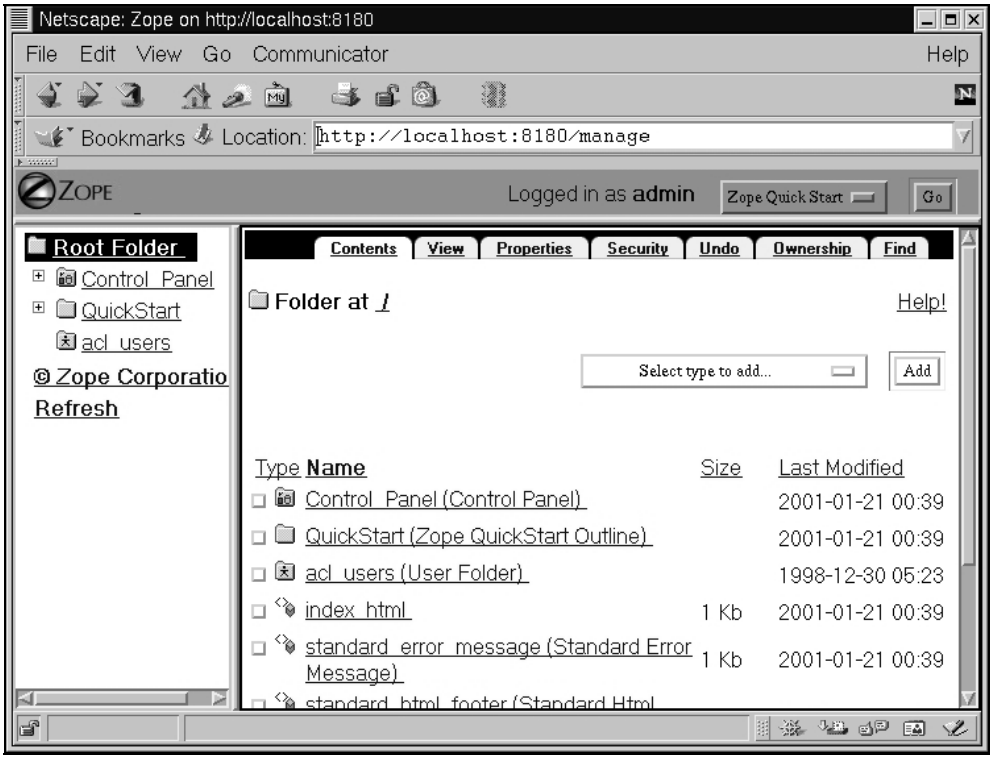


Рис. 14.1. Страница управления Zope

14.1.2. Политика безопасности

Прежде чем приступать к созданию ресурсов и пользователей, необходимо познакомиться с моделью безопасности, действующей в Zope. Управление безопасностью сервера Zope необходимо вести правильно с самого начала. Нужно точно знать, кто и что может делать на сервере помимо администратора (который может все).

Прежде чем управлять содержанием или пользоваться закрытыми ресурсами, пользователь должен пройти *аутентификацию* и *авторизацию*. При аутентификации выясняется, кто именно хочет получить доступ. После этого сервер проверяет, авторизован ли пользователь на доступ к ресурсу, который он запрашивает.

Для упрощения авторизации пользователям Zope могут быть приданы определенные *роли (roles)*: `Owner` (владелец), `Manager` (управляющий), `Anonymous` (безымянный пользователь) и т. п. С другой стороны, объекты требуют наличия *прав (permissions)* на выполнение определенных операций: `View` (про-

смотр), `Add folders` (добавление папки), `Manage users` (управление пользователями) и многие другие. Для каждого ресурса администратор может задать *политику безопасности (security policy)* — матрицу, в которой указано, какие операции могут делать пользователи в определенных ролях. Матрицу можно заполнить, обратившись к вкладке `Security` в главном фрейме. Права автоматически распространяются на нижележащие папки.

Как уже было сказано, к конкретной папке можно добавить список пользователей, которые имеют право работать в этой папке (и нижележащих папках). Эти пользователи не имеют доступа в вышележащие папки.

Таким образом, Zope позволяет достаточно просто управлять доступом различных групп пользователей и разработчиков к Web-сайту и делегировать управление отдельными разделами конкретным пользователям-управляющим.

14.1.3. Добавляем ресурс и пользователей

Предположим, что мы хотим открыть новый проект, который будет находиться в папке `Example`. Для этого нужно просто выйти в корневую папку (щелкнув на **Root Folder** в левом фрейме) и выбрать в главном фрейме из меню **Select type to add...** пункт **Folder**. Требуется набрать внутреннее имя-идентификатор (например, `Example`, в имени не должно быть пробелов) и название (например, `Example Project`). Кроме того, можно поставить "галочки" в обоих флажках: **Create user folder** (Создать папку со списком пользователей) и **Create public interface** (Создать внешний интерфейс). После нажатия на кнопку **Add** папка появится в списке. Чтобы она появилась и в левом навигационном фрейме, нужно щелкнуть по ссылке **Refresh** (Обновить).

Теперь можно наполнить список пользователей проекта. Войдите в папку `Example`, а затем откройте файл acl_users. Добавить пользователя можно кнопкой **Add**. Для пользователя требуется ввести его имя, пароль (дважды) и назначить ему одну или несколько ролей. По умолчанию доступны роли `Owner` (владелец) и `Manager` (управляющий). Если роли не определены, будет считаться, что пользователь имеет роль `Authenticated` (авторизованный, небезымянный). Разумеется, для работы над конкретным проектом можно добавить несколько пользователей.

Содержимое (баннеры) будут находиться в папке `Example/bans`. Создайте эту папку, не забыв отметить пункт **Create user folder**. Для нашего примера нужно завести пользователя `clerk` и присвоить ему роль `banner_manager`. Но до этого нужно определить саму роль `clerk`, разрешив ей только доступ к содержимому (Access contents information), добавление документов, изображений и файлов (Add Documents, Images, and Files), изменение изображений и файлов (Change Images and Files), удаление объектов (Delete objects), отмену изменений (Undo changes), просмотр (View) и доступ к интерфейсу управления (View management screens).

14.2. С точки зрения программиста

Наверняка решаемая вами задача не является стандартной и без помощи программиста вам просто не обойтись. Что же дает программисту Zope?

14.2.1. Zope-объекты

Прежде всего, Zope является *системой публикации объектов (object publishing system)*. В Zope объекты являются кирпичиками для построения приложений и отвечают за их логику, представление и содержание.

Одним из основополагающих объектов в Zope является *папка (Folder)*. Папка является контейнером для других объектов. Как и в современных файловых системах, папки могут быть вложенными. Например, все папки Zope в одной корневой папке (Root Folder). Все остальные объекты прямо или опосредованно находятся в этой папке.

В табл. 14.1 перечислены некоторые объекты, которые доступны сразу после установки Zope.

Таблица 14.1. *Некоторые объекты Zope*

Объект	Описание
Accelerated HTTP Cache Manager	Ускоренный менеджер кэша HTTP
DTML Document	Документ на DTML (Document Template Markup Language, язык разметки шаблонов документов). Является основным средством для представления информации конечному пользователю Web-приложения. Позволяет писать шаблоны не только для HTML, но и XML. В DTML-документе можно использовать другие Zope-объекты
DTML Method	Шаблон DTML. Может использоваться для представления отдельных частей DTML-документа. Например, для представления одного и того же содержимого может использоваться несколько различных шаблонов
External Method	Внешний метод. Соответствует сценарию, записанному вне файловой системы Zope, в файловой системе хоста, на котором установлен Zope. Внешние сценарии используются там, где не обойтись без кода, который обходит обычные в Zope ограничения безопасности. Применяются для доступа к библиотекам (недоступным из Zope), файлам из файловой системы хоста и т. п.
File	Файл. Может содержать любые данные, как и файл в обычной файловой системе
Folder	Папка. Контейнер для Zope-объектов
Image	Изображение. Аналогичен файлу, но содержит дополнительные возможности, специфичные для изображений

Таблица 14.1 *(окончание)*

Объект	Описание
Mail Host	Объект для отправки почты из Zope
RAM Cache Manager	Кэш в оперативной памяти. Позволяет кэшировать в оперативной памяти документы из папки для оптимизации доступа
Script (Python)	Сценарий (на Python). Является рекомендуемым способом для программирования логики приложения. Начиная с Zope 2.3, сценарии можно писать и на языке Perl
User Folder	Папка с описаниями пользователей
Version	Контроль версий
User Folder	Папка с описаниями пользователей
Version	Контроль версий
Z Gadfly Database Connection	Соединение со встроенной в Zope (демонстрационной) базой данных Gadfly. Для связи с другими базами данных необходимо устанавливать соответствующие адаптеры, например ZMySQLDA
Z SQL Method	Объект, реализующий SQL-запросы к базе данных
ZCatalog	Объект для организации поиска объектов по различным критериям

14.2.2. Знакомимся с языком DTML

Ниже мы рассмотрим пример, в котором окончательный HTML-документ будет создаваться на основе DTML-документа и Python-сценария. Работать мы будем в папке `Example`, созданной администратором в *разд. 14.1.3*.

Суть задачи состоит в следующем. В нижележащей папке `Example/bans` будут храниться баннеры, которые должны появляться в намеченных местах. Количество баннеров может изменяться (их добавлением, изменением и удалением будет заниматься уже другой человек — об этом в следующем разделе).

Для начала немного познакомимся с языком DTML. В листинге 14.1 выводится информация обо всех изображениях (объектах `Image`) из папки `bans`.

Листинг 14.1. Файл list_all_images. Пример DTML-разметки (метод `list_all_images`)

```
<html><head><title><dtml-var title></title></head>
<body>
<h2><dtml-var title></h2>
<dtml-if expr="bans.objectIds('Image')">
 <table border=1>
```

```
 <tr><th>N</th><th>Id</th><th>Изображение</th><th>Размер</th></tr>
 <dtml-in expr="bans.objectItems('Image')">
  <dtml-let i=sequence-item>
  <tr><th><dtml-var sequence-Roman></th>
  <td><dtml-var expr="i.title_or_id()"></td>
  <td><dtml-var i></td>
  <td><dtml-var expr="i.width"> x <dtml-var expr="i.height"></td></tr>
  </dtml-let>
 </dtml-in>
 </table>
<dtml-else>
Изображений нет.
</dtml-if>
</body></html>
```

На рис. 14.2 приведен результат работы DTML-метода `list_all_images`.



Рис. 14.2. Результат работы метода `list_all_images`

Наверное, вы уже заметили, что все теги, относящиеся к DTML, начинаются с "<dtml-". После этого следует одно из ключевых слов, которое и определяет DTML-команду. В табл. 14.2 приведены некоторые из тегов DTML. Часть тегов должна быть легко понятна тем, кто знаком с языком Python: это аналоги операторов Python.

Таблица 14.2. *Некоторые теги DTML*

Теги	Описание
`<dtml-var объект>`	Вставка обозначенного объекта
`<dtml-call объект>`	Вызов объекта
`<dtml-with пространство_имен [only]>, </dtml-with>`	Позволяет использовать `пространство_имен` прежде всех остальных в ограниченной тегом области. Если задан атрибут `only`, то внутри можно использовать только это пространство имен
`<dtml-let имя="значение">, </dtml-let>`	Вводит имена, которые действуют внутри ограниченной тегом области (похож на тег `<dtml-with>`, но позволяет вводить сразу несколько имен)
`<dtml-if условие>, <dtml-elif>, <dtml-else>, </dtml-if>`	Аналог оператора `if-elif-else`
`<dtml-unless условие>, </dtml-unless>`	Аналогичен `if not условие`
`<dtml-in>, </dtml-in>`	Аналогичен оператору `for`. В отличие от него, конкретный параметр цикла не используется — вместо него внутри цикла можно применять переменную `sequence-item` или, если требуется не сам объект, а его индекс, переменную `sequence-index` и ее разновидности
`<dtml-comment>, </dtml-comment>`	Комментарий. В отличие от HTML-комментария, клиенту не передается
`<dtml-try>, <dtml-except>, <dtml-finally>, </dtml-try>`	Аналог операторов
`<dtml-raise type="тип">, </dtml-raise>`	Возбуждает исключение указанного типа (аналогично `raise` в Python)
`<dtml-sendmail>, </dtml-sendmail>`	Команда для форматирования и отправки письма
`<dtml-mime type="тип" encode="кодировка">, </dtml-mime>`	Команда для форматирования MIME-сообщения. Например, вложения

Таблица 14.2 *(окончание)*

Теги	Описание
`<dtml-tree>, </dtml-tree>`	Команда, позволяющая обходить дерево вложенных друг в друга объектов

Примечание

Кое-где вы можете встретить и более старый вариант разметки, использующий HTML-комментарии. Например: "`<!--#var title-->`".

Теперь нам будет легко разобраться с тем, что же делает DTML-метод `list_all_images`. Переменная `title` (название) является атрибутом объекта `list_all_images` (такой атрибут, а также атрибут `id` имеют практически все Zope-объекты). В нашем примере `title` имел значение "`Все баннеры`", поэтому в результате двух применений команды `dtml-var` это значение оказалось как в названии HTML-страницы, так и в заголовке (см. рис. 14.2).

Следующая конструкция:

```
<dtml-if expr="bans.objectIds('Image')">...<dtml-else>...</dtml-if>
```

позволяет узнать, есть ли объекты типа `'Image'` в папке `bans`. Если таковые имеются, выполняется первая часть (до `<dtml-else>`), если нет — вторая, в которой на HTML-страницу попадает надпись "`Изображений нет.`". В первой части строится таблица, в строках которой размещается информация об изображениях в папке `bans`:

```
<dtml-in expr="bans.objectItems('Image')">...</dtml-in>
```

Здесь задан цикл по всем элементам указанного списка объектов. При каждом проходе цикла создается одна строка таблицы. Командой

```
<dtml-let i=sequence-item>
```

мы присваиваем переменной `i` значение текущего элемента. Теперь мы можем вместо `<dtml-var sequence-item>` писать `<dtml-var i>`, а также применять `i` в выражениях. В первом столбце таблицы помещается номер изображения (римскими цифрами). Это достигается применением команды:

```
<dtml-var sequence-Roman>
```

В следующем столбце выводится название или идентификатор изображения (результат работы метода `title_or_id()` объекта `i`):

```
<dtml-var expr="i.title_or_id()">
```

В качестве значения атрибута `expr` можно применять выражения на языке Python, правда, с некоторыми ограничениями. Например, доступные в Zope

встроенные функции Python приходится задавать через специальное пространство имен "_":

```
<dtml-in expr="_.range(1, 11)"><br><dtml-var sequence-item></dtml-in>
```

В третьем столбце выводится само изображение, а в четвертом — его ширина и высота, получаемые из соответствующих атрибутов объекта-изображения.

Замечание

Не следует слишком увлекаться программированием на DTML. Этот язык хорош для создания шаблонов документов, а не для программирования. Для нетривиальных действий в Zope лучше использовать сценарии, которые пишутся на языке Python или Perl.

В следующем разделе мы будем писать сценарий на языке Python для показа произвольного баннера из папки `bans`.

14.2.3. Пишем сценарий

Zope-сценарий — объект, содержащий небольшой фрагмент кода на языке программирования, — позволяет программировать логику Web-приложения более эффективно, чем DTML-метод. На момент написания книги Zope-сценарии можно писать как на Python, так и на Perl.

В отличие от "обычных" сценариев на Python, Zope снабжает сценарии некоторыми специфичными для Web-приложения объектами (табл. 14.3). Эти объекты говорят сценарию об окружении, в котором он исполняется, и могут иметь значение `None`, если в конкретной ситуации объект отсутствует (не имеет смысла). Кроме того, сценарию могут быть переданы параметры (как через обычные параметры CGI-формы, так и при вызове из других сценариев). Параметры доступны в сценарии под своими именами.

Таблица 14.3. *Некоторые предопределенные имена в Zope-сценарии*

Объект	Типичное имя в сценарии	Назначение
`context`	`context`	Объект, для которого сценарий вызван
`container`	`container`	Объект, содержащий сценарий (обычно это папка или `ZClass`)
	`container.REQUEST`	Объект класса `Request`, из которого можно получить информацию из запроса, например, значения ключиков (cookies)

Таблица 14.3 *(окончание)*

Объект	Типичное имя в сценарии	Назначение
	`container.URLn`	URL запроса (без строки запроса). Если задано `n`, URL возвращается без `n` последних компонентов. `container.URL0` тоже самое, что и `container.URL`
	`container.REQUEST.RESPONSE`	Объект класса `Response`, содержащий ответ на запрос. Через него можно, например, задавать тип возвращаемого документа или устанавливать ключики (cookies)
`script`	`script`	Сам объект-сценарий
`namespace`	`_`	Пространство имен вызывающего скрипт объекта. Например, при вызове из DTML-документа это может быть "_"
`subpath`	`traverse_subpath`	При публикации сценария напрямую с помощью URL содержит список компонентов пути после имени сценария в URL. Является пустым, если имя сценария — последний компонент URL (не считая строки запроса)

Примечание

Соответствие между объектами и их именами можно настроить индивидуально для каждого сценария.

После этих объяснений можно рассмотреть пример простейшего сценария для выбора случайного объекта-изображения (листинг 14.2).

Листинг 14.2. Файл any_banner. Метод для показа случайного баннера

```
import random
# Выбрать случайный элемент списка объектов типа 'Image'
# из папки bans, находящейся в папке, откуда вызван сценарий.
return random.choice(context.bans.objectValues('Image'))
```

Чтобы опробовать сценарий, можно включить в метод `index_html` следующее:

```
Любой баннер: <dtml-var any_banner>
```

Теперь все, кто выходит на Zope-сервер в каталог `Example`, увидят среди прочего произвольно выбранный баннер из папки `bans`.

Мы работаем в папке Example, но ничто не мешает нам создать в этой папке другие папки и в них папку bans. И теперь самое интересное: сценарий any_banner будет действовать в любой подпапке (даже глубоко вложенной) из папки Example. Эта особенность объектной модели Zope называется *приобретение (acquisition) объектов*. Приобретение является одним из основных механизмов повторного использования объектов и их свойств. Даже настройки из раздела Security приобретаются из вышележащих папок, а после приобретения некоторых прав в данной папке пользователь по умолчанию приобретает права на все объекты, расположенные в подпапках. Таким образом, приобретение (acquisition) является очень важной концепцией Zope, без понимания которой его сложно грамотно применять.

Следующий пример показывает, что приобретение (acquisition) можно использовать и для папок. Заведем папку Zigzag на том же уровне, что и папка Example, и перенесем в нее папку bans (то есть теперь в Example папки bans быть не должно). Теперь с помощью URL

```
http://zopeserver/Zigzag/Example/index_html
```

мы увидим баннер из папки Zigzag/bans! Весь секрет состоит в том, что метод index_html и сценарий any_banner приобрели папку bans из папки Zigzag (так как в папке Example ее не было). Таким образом, при помощи "хитрого" URL можно добиться требуемого пути приобретения объектов. Из приведенного примера можно также заметить, что Zope-сервер трактует URL несколько иначе чем, скажем, обычный Web-сервер. В Zope URL может делать некоторые вполне осмысленные зигзаги.

Теперь несколько усложним наш сценарий, добавив к нему параметр. Пусть нам необходимо показывать баннеры только одного размера. Этот размер мы будем передавать в параметре size. В листинге 14.3 приведен сценарий, выбирающий из папки баннеры заданного размера.

Листинг 14.3. Файл any_banner2. Метод для показа случайного баннера (вариант 2)

```
import random
list_of_banners = []
for banner in context.bans.objectValues('Image'):
  if size == "%sx%s" % (banner.width, banner.height):
    list_of_banners.append(banner)
if list_of_banners:
  return random.choice(list_of_banners)
else:
  return ''
```

Вызвать этот сценарий из DTML можно, к примеру, так:

```
<dtml-var expr="any_banner2('140x64')">
```

14.3. С точки зрения менеджера по содержанию

Содержание (content) совсем необязательно управляется администратором или программистом: даже в небольших проектах за это, как правило, отвечает отдельный человек. Соответственно, права этого специалиста в системе Zope ограничиваются возможностью манипулировать определенными видами объектов.

Заведенный администратором пользователь с именем `clerk` может выйти на интерфейс управления в папку `bans` по адресу **http://zope-server/Example/bans/manage** (рис. 14.3).

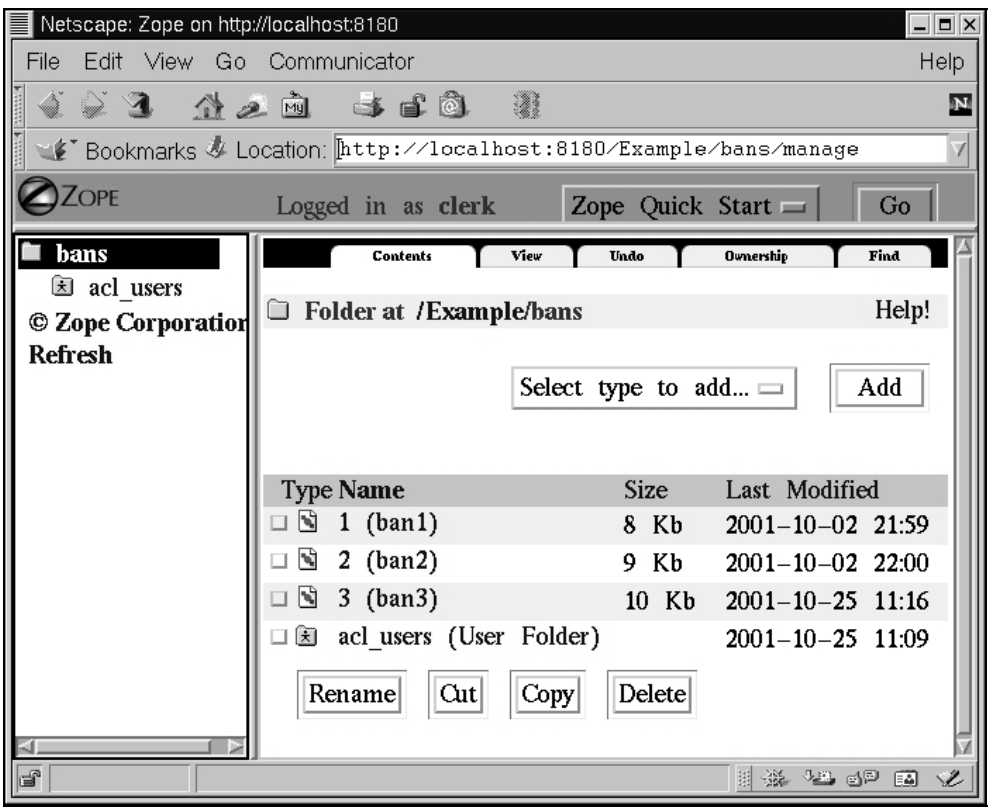


Рис. 14.3. Папка bans

Для помещения в папку нового баннера нужно выбрать из меню **Select type to add...** тип `Image` и заполнить форму (рис. 14.4).

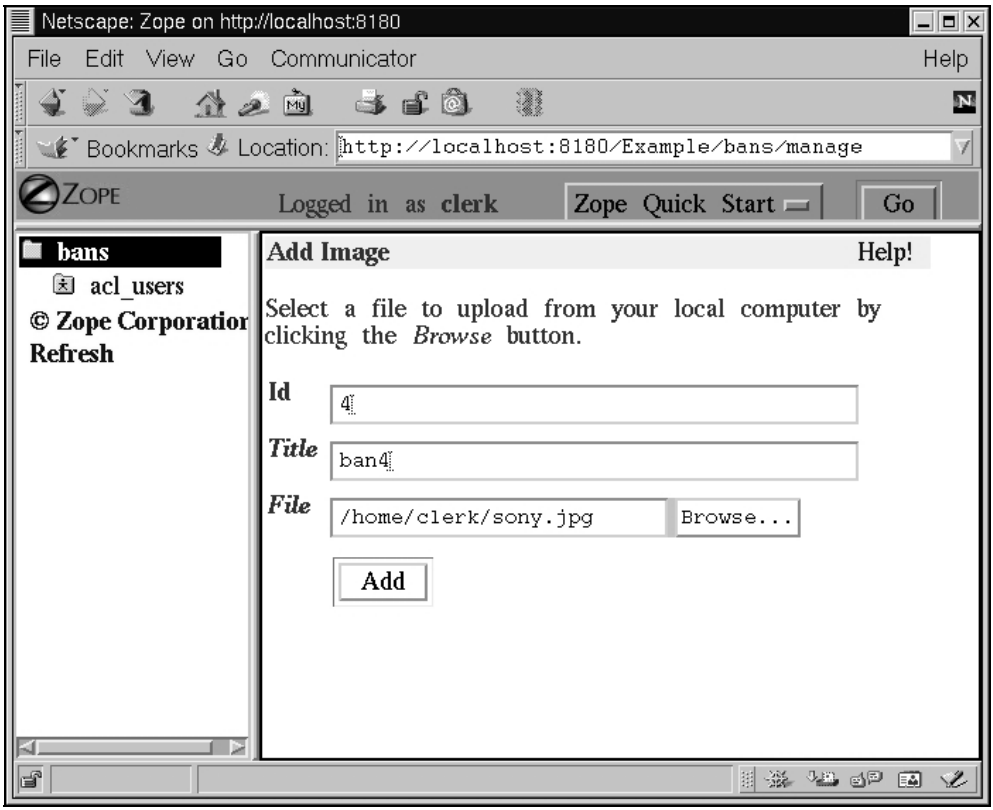


Рис. 14.4. Загрузка нового баннера в папку `bans`

Конечно, программисты могли бы легко создать и другой интерфейс для управления содержанием, но для нашей задачи достаточно стандартного интерфейса управления Zope. В табл. 14.4 приведено назначение кнопок, присутствующих в этом интерфейсе. Выбрать объекты можно с помощью флажков, которые присутствуют в каждой строке.

Таблица 14.4. *Назначение кнопок интерфейса управления Zope*

Кнопка	Перевод	Описание
Rename	Переименовать	Запрашивает новые имена (Id) для выбранных объектов

Таблица 14.4 *(окончание)*

Кнопка	Перевод	Описание
Cut	Вырезать	Переносит в буфер выбранные объекты. В данной папке объект остается только до вставки его в другую папку
Copy	Скопировать	Копирует выбранные объекты в буфер
Paste	Вставить	Вставляет в папку ранее сохраненные в буфере объекты
Delete	Удалить	Удаляет выбранные объекты из папки
Import/Export	Импортировать/экспортировать	Позволяет сохранить объекты в файле или считать их из файла вне Zope-сервера
Select All, **Deselect All**	Выбрать все, сбросить выбор	Служит для выбора всех объектов или снятия выбора

Щелкнув мышью на названии объекта, можно попасть на страницу конкретного объекта, вкладку **Edit**. Здесь приведены свойства объекта (в нашем примере — файла с изображением баннера), и предоставляется возможность загрузить другой файл с изображением с помощью кнопок **Browse** (Просмотр) и **Upload** (Загрузить). Чтобы посмотреть изображение баннера, нужно выбрать вкладку **View**. Вкладка **Properties** открывает страницу для задания свойств объекта. Здесь, в частности, можно задать не только стандартные свойства, но и другие. Например, многие баннеры имеют ссылку на рекламируемый сайт и могут иметь определенное значение приоритета при показе (баннеры с большим приоритетом будут показываться чаще баннеров с меньшим приоритетом), и т. п. (В нашем примере дополнительные свойства не используются.)

Глава 15



Jython — Python на Java

Jython, известный до этого как JPython, является реализацией современных версий языка Python для виртуальной Java-машины. Под JPython теперь понимается реализация версий 1.5.2, 1.6 и т. д. Jython может свободно использоваться как в некоммерческих, так и в коммерческих проектах. Он позволяет использовать всю мощь библиотек Java из кода на Python, т. к. полностью интегрирован в Java-платформу. С помощью Jython программисты на Java могут встраивать Python-программы в код на Java, использовать интерактивный режим интерпретатора Jython для работы с Java-системой. Можно легко заметить, что программы на Python обычно короче соответствующих программ на Java в несколько раз, что позволяет существенно сократить сроки разработки как прототипов, так и готовых программных решений.

В документации по Jython с гордостью отмечено, что, в отличие от других языков на платформе Java, Jython обладает следующими неоспоримыми преимуществами:

- Jython-код динамически компилирует байт-коды Java, хотя возможна и статическая компиляция, что позволяет писать апплеты, сервлеты и т. п.;
- Jython поддерживает объектно-ориентированную модель Java, в том числе, возможность наследовать от абстрактных Java-классов;
- Jython является реализацией Python — языка с практичным синтаксисом и обладающего в то же время большой выразительностью.

Ниже "обычная" реализация Python, выполненная на C, будет называться CPython, а под Python будет пониматься Python как язык. Кстати, Jython — не единственная альтернативная реализация Python. Есть еще Stackless Python (бесстековый Python), а также Parrot (смесь Perl и Python, если разработчики не шутят).

Конечно, некоторые возможности Jython по сравнению с Python ограничены. Например, Java не поддерживает множественного наследования, поэтому в некоторых версиях Jython нельзя наследовать классы от нескольких Java-классов (в то же время, множественное наследование поддерживается для Jython-классов).

Между CPython и Jython есть и другие различия. В основном они достаточно мелкие, но игнорировать их совсем не удастся. Следующий список не претендует на полноту, однако дает хорошее представление, чем же отличается Jython от Python.

- Jython позволяет записывать литералы чисел с плавающей запятой с нулями в начале, например: `009.23`;
- все строки Jython являются Unicode-строками со всеми вытекающими отсюда последствиями;
- Jython использует для получения свойств Unicode-символов методы языка Java, базирующиеся на Unicode-2, тогда как CPython поддерживает Unicode-3 и потому имеет бóльшие возможности. Речь идет о таких свойствах, как регистр букв, признаках того, что символ является буквой, и т. п.;
- каждый объект в Jython является экземпляром класса. Таким образом, в Jython нет типов;
- имеются некоторые отличия в работе с файловыми объектами;
- в Jython используется сборка мусора, а не комбинация сборки мусора и счетчика ссылок, как в CPython. С одной стороны, это избавляет от необходимости следить за циклическими ссылками, с другой — в Jython сложно угадать, когда именно объект прекратит свое существование, что несколько неудобно для тех, кто использует конструкции типа: `file_data = open("file").read()`;
- разумеется, результаты интроспекции CPython и Jython могут отличаться. Даже обычный `type(1)` дает разные строки в CPython и Jython;
- имеется разница в составе и наполнении стандартных модулей. Здесь следует отметить, что в Jython можно подгружать пакеты и модули Java с помощью обычного оператора импорта;
- в Jython отсутствуют функции типа `os.popen` или `os.system`, ведь Jython ориентирован на Java-платформу, а CPython — на POSIX.

Конечно, есть много других мелких различий между Jython и CPython. Однако следует заметить, что интерпретатор может называться реализацией Python, не являясь стопроцентным аналогом CPython. Документация по языку Python сознательно умалчивает о некоторых деталях или особенностях реализации, а в других случаях прямо предупреждает о возможных вариантах. Это позволяет Python быть самим собой в самых разных перерождениях (инкарнациях).

Самое интересное в Jython, наверное, возможность прямого использования модулей Java, что и показано в листинге 15.1.

Листинг 15.1. Файл draw_lines.py. Пример использования модулей Java в Jython: рисование линий

```
#!/usr/bin/env jython

from java.lang import System
from java.awt import *
import random

class Lines(Canvas):
  def paint(self, g):
    X, Y = self.getSize().width, self.getSize().height
    label.setText("%s x %s" % (X, Y))
    for i in range(100):
      x1, y1 = random.randint(1, X), random.randint(1, Y)
      x2, y2 = random.randint(1, X), random.randint(1, Y)
      g.drawLine(x1, y1, x2, y2)

panel = Panel(layout=BorderLayout())
label = Label("Size", Label.RIGHT)
panel.add(label, "North")
button = Button("QUIT", actionPerformed=lambda e: System.exit(0))
panel.add(button, "South")
lines = Lines()
panel.add(lines, 'Center')

import pawt
pawt.test(panel, size=(240, 240))
```

Программа формирует панель с текстовой этикеткой `label`, кнопкой **QUIT** (которая при нажатии на нее вызывает указанную заданной функцией команду) и рисунком, которым управляет класс `Lines`. В этом классе, наследованном от Java-класса `java.awt.Canvas`, определен всего один метод — `paint()`. Этот метод используется для генерации изображения рисунка и вызывается при необходимости перерисовки рисунка. Модуль `pawt` (из комплекта `Jython`) несет вспомогательную функцию и позволяет быстро создать окно нужного размера. Результат работы программы можно увидеть на рис. 15.1. Стоит отметить следующую интересную особенность: метод `paint()` вызывается при необходимости любой перерисовки прямоугольни-

ка с рисунком, даже если это небольшая прямоугольная полоска. И каждый раз будут нарисованы разные линии!

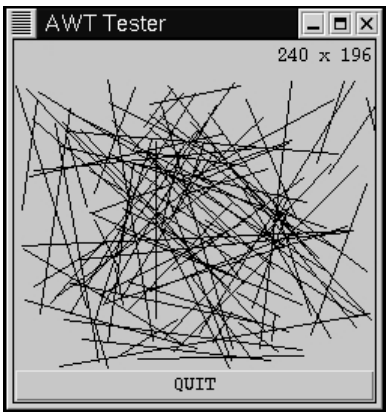


Рис. 15.1. Результат работы программы `draw_lines.py`

В примере использован пакет `java.lang.awt`, однако из Jython можно применить практически любой другой доступный в Java модуль.

Программы на Jython можно компилировать в Java и собирать, например, в JAR-архив. Для создания JAR-архива на основе модуля (или серии модулей) можно применить `jythonc`, входящий в состав поставки Jython. Вот как это делается из командной строки:

```
jythonc -c -d -j dl.jar draw_lines.py
```

где `-c` — опция, заставляющая `jythonc.py` помещать в JAR-архив все необходимые модули, включая Jython; `-d` — опция для включения всех зависимостей данного модуля; `-j` — опция для указания названия JAR-архива;

Заметьте, что в ходе работы `jythonc` создает каталог `jpywork`, в котором можно найти java-файлы, соответствующие py-файлам. Для запуска приложения достаточно запустить `draw_lines`:

```
java -classpath "$CLASSPATH" draw_lines
```

Вместо `$CLASSPATH` нужно указать пути к архивам `dl.jar` и `jython.jar`.

Аналогично можно создать апплет, написанный на Jython. В листинге 15.2 приведен апплет, сделанный на основе программы `draw_lines.py`. Важно, чтобы название класса, который будет считаться апплетом, совпадало с именем модуля. В нашем примере это — `draw_lines_app`. Из примера убрана кнопка (апплету выход не требуется), а также конструктор класса `Lines` принимает параметр `label`, который ранее был глобальной переменной. Кроме того, вместо панели `panel` используется сам апплет.

Листинг 15.2. Файл draw_lines_app.py. Апплет на Jython

```
from java.awt import *
from java.applet import Applet
import random

class Lines(Canvas):
  def __init__(self, label):
    self.label = label
  def paint(self, g):
    X = self.getSize().width
    Y = self.getSize().height
    self.label.setText("%s x %s" % (X, Y))
    for i in range(100):
      x1 = random.randint(1, X)
      y1 = random.randint(1, Y)
      x2 = random.randint(1, X)
      y2 = random.randint(1, Y)
      g.drawLine(x1, y1, x2, y2)

class draw_lines_app(Applet):
  def __init__(self):
    self.setLayout(BorderLayout())
    label = Label("Size", Label.RIGHT)
    self.add(label, "North")
    self.lines = Lines(label)
    self.add(self.lines, 'Center')
```

Как и для примера выше создаем JAR-архив:

```
jythonc -c -d -j drawlines.jar draw_lines_app.py
```

Чтобы увидеть работу апплета, нужно поместить его на HTML-страницу. Пример такой страницы дан в листинге 15.3. Вместе с этой страницей должны лежать JAR-архивы Jython и апплета. На странице апплет получает конкретные размеры — ширину `width` и высоту `height`, заданные параметрами в теге `<applet>`.

Листинг 15.3. Файл app.html. HTML-страница с апплетом

```
<html><head>
   <title>Пример апплета на Jython</title>
```

```
</head><body>
<applet code="draw_lines_app.class"
      archive="drawlines.jar, jython.jar"
      width=200 height=200>
</applet>
</body></html>
```

Интерпретатор Python можно встраивать в программы на Java. В листинге 15.4 приведен пример программы на Java, вызывающей Python.

Листинг 15.4. Файл PythonInside.java. Python, встроенный в Java

```
import org.python.util.PythonInterpreter;
import org.python.core.*;
import java.util.*;

public class PythonInside {
  public static void main(String []args) throws PyException {
    /* Получаем экземпляр интерпретатора */
    PythonInterpreter python = new PythonInterpreter();
    /* Что-то делаем на Java */
    System.out.println("I am Java!");
    /* Пробуем Python */
    python.exec("print 'I am Python!'");
    /* Пробуем многострочный кусок программы */
    python.exec("print '*'*10           \n" +
                "for i in range(5):     \n" +
                "  print '*'*i            ");
    /* Пример с передачей данных в Python и обратно. */
    /* Присваиваем значение строке */
    python.set("s", new PyString("One Two Three"));
    /* Выполняем разбивку строки (в Python) */
    python.exec("s1 = s.split()");
    /* Печатаем результат вычисления выражения s1[0] */
    System.out.println("s1[0]: " + python.eval("s1[0]"));
    /* Получаем из интерпретатора Python-объект s1 */
    PyObject s1 = python.get("s1");
    /* Печатаем объект s1 */
    System.out.println("s1: " + s1);
  }
}
```

Чтобы попробовать приведенную программу, нужно сначала ее откомпилировать, а затем запустить:

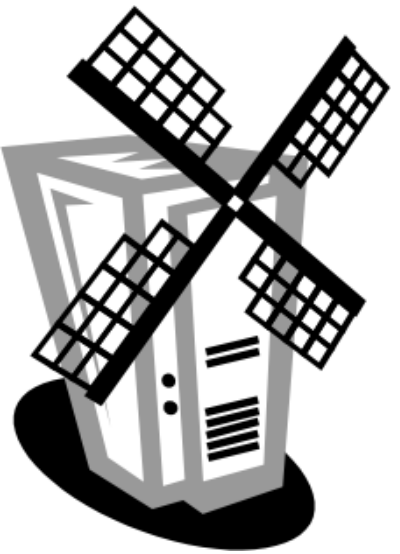
```
javac -classpath $CLASSPATH SimpleEmbedded.java
java -classpath $CLASSPATH SimpleEmbedded
```

Здесь вместо `$CLASSPATH` нужно указать текущий каталог и путь к JAR-архиву `jython.jar`. А вот что получается в результате исполнения программы:

```
I am Java!
I am Python!
**********

*
**
***
****
s1[0]: One
s1: ['One', 'Two', 'Three']
```

Конечно, `PyObject` имеет специальные методы для согласования с различными Java-классами, поэтому если вам необходим Jython, изучите документацию к нему. Отправной точкой может служить **http://www.jython.org**.



Часть IV

Приложения

Приложение 1. Переводы сообщений об ошибках

Приложение 2. Оболочка IDLE

Приложение 3. Описание компакт-диска

Приложение 1



Переводы сообщений об ошибках

В табл. П1.1 приведены переводы сообщений об ошибках, наиболее часто встречающиеся при работе с интерпретатором Python.

Таблица П1.1. *Переводы некоторых сообщений об ошибках*

Тип ошибки	Ошибка	Перевод
`AttributeError`	class X has no attribute Y	Класс `X` не имеет атрибута `Y`
`AttributeError`	instance has no attribute Y	Экземпляр не имеет атрибута `Y`
`AttributeError`	module has no attribute Y	Модуль не имеет атрибута `Y`
`AttributeError`	object has no attribute Y	Объект не имеет атрибута `Y`
`IOError`	Cannot open file	Не могу открыть файл
`IndexError`	... index out of range	Индекс за пределами допустимого диапазона
`IndexError`	pop from empty list	Попытка применить `pop()` к пустому списку
`MemoryError`	stack overflow	Переполнение стека
`NameError`	name 'X' is not defined	Имя `X` не определено
`OverflowError`	cannot add more objects to list	Не могу добавить элементы к списку
`OverflowError`	formatted X is too long (precision too large?)	Форматированное число `X` слишком длинное. Задана слишком большая точность?
`OverflowError`	long int too large to convert	Длинное целое слишком велико для преобразования
`OverflowError`	repeated string is too long	Повторение породит слишком длинную строку
`OverflowError`	string is too long for a Python string	Строка слишком велика

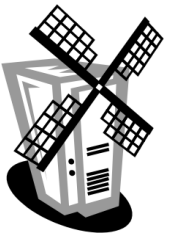
Таблица П1.1 *(продолжение)*

Тип ошибки	Ошибка	Перевод
`OverflowError`	xrange object has too many items	Слишком большой `xrange`
`SyntaxError`	invalid syntax	Неверный синтаксис
`SyntaxError`	non-keyword arg after keyword arg	Неименованный аргумент вслед за именованным
`TypeError`	can't convert X to Y	Не могу преобразовать `X` в `Y`
`TypeError`	'in <string>' requires character as left operand	Слева должна стоять односимвольная строка
`TypeError`	'in' or 'not in' needs sequence right argument	`in` или `not in` требует последовательность справа
`TypeError`	F() got an unexpected keyword argument 'X'	X не может быть именованным аргументом F
`TypeError`	F() takes at least N arguments (M given)	F принимает не менее N аргументов (дано M)
`TypeError`	F() takes at most N arguments (M given)	F принимает не более N аргументов (дано M)
`TypeError`	F() takes exactly N arguments (M given)	F принимает точно N аргументов (дано M)
`TypeError`	__repr__ returned non-string (type X)	`__repr__()` возвратил не строку, а X
`TypeError`	__str__ returned non-string (type X)	`__str__()` возвратил не строку, а X
`TypeError`	an integer is required	Требуется целое число
`TypeError`	argument has non-string str()	`str()` от аргумента не дает строку
`TypeError`	attribute name must be string	Имя атрибута должно быть строкой
`TypeError`	attribute-less object (assign or del)	Попытка удалить или присвоить атрибут объекта, не имеющего атрибутов
`TypeError`	expected string without null bytes	Ожидалась строка без нулевых битов
`TypeError`	format requires a mapping	Операция форматирования требует отображения (например, словаря)
`TypeError`	not all arguments converted	Не все аргументы использованы (при форматировании)

Таблица П1.1 *(окончание)*

Тип ошибки	Ошибка	Перевод
`TypeError`	not enough arguments for format string	Недостаточно аргументов для форматирования
`TypeError`	number coercion failed	Согласование типа чисел невозможно
`TypeError`	read-only attribute: X	Атрибут только для чтения
`TypeError`	read-only special attribute	Специальный атрибут только для чтения
`TypeError`	sequence expected, X found	Ожидалась последовательность, найдено `X`
`TypeError`	this constructor takes no arguments	Конструктор не принимает аргументов
`TypeError`	unbound method X must be called with instance as first argument	Несвязанный метод должен вызываться с экземпляром класса в качестве первого аргумента
`TypeError`	unhashable instance	От экземпляра нельзя получить хэш-значение
`TypeError`	unsupported operand type(s) for A	Неподдерживаемый тип операнда для `A`
`UnboundLocalError`	local variable 'X' referenced before assignment	Ссылка на переменную `X` до присваивания
`UnicodeError`	ASCII encoding error: ordinal not in range(128)	Код не лежит в диапазоне 0–127
`ValueError`	I/O operation on closed file	Попытка операции ввода/вывода на закрытом файле
`ValueError`	empty pattern string	Строка с образцом пуста
`ValueError`	incomplete format	Неполный формат
`ValueError`	incomplete format key	Неполный ключ форматирования
`ValueError`	not in list	Значение не в списке
`ValueError`	substring not found in string.index	Подстрока не найдена
`ValueError`	translation table must be 256 characters long	Строка преобразования должна иметь длину 256 символов
`ValueError`	unsupported format character X	Неподдерживаемый спецификатор формата `X`
`ValueError`	width too big	Слишком большая ширина

Приложение 2



Оболочка IDLE

Для тех, кому удобнее работать в интегрированной среде разработчика, Python может предоставить *оболочку IDLE*. IDLE дает следующие возможности:

- интерактивная оболочка Python с выделением синтаксиса цветом
- специализированный текстовый редактор (тоже с выделением синтаксиса)
- просмотр путей поиска модулей
- просмотр классов модуля
- отладка модуля

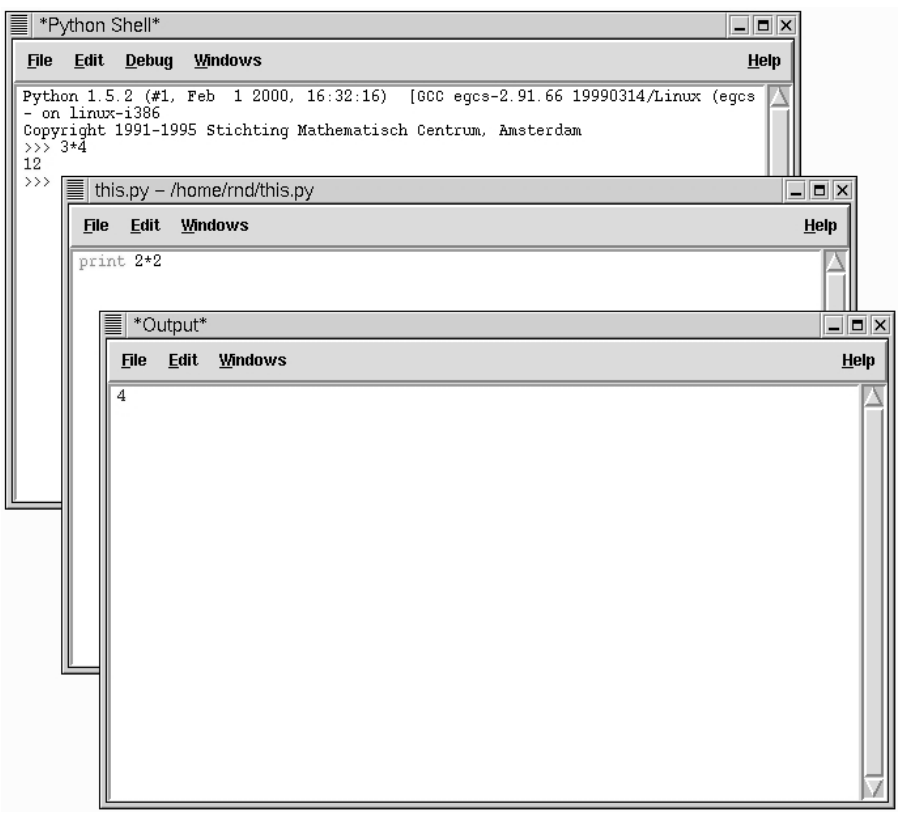


Рис. П2.1. Основные типы окон в IDLE

IDLE работает с несколькими видами окон: окна с интерактивным приглашением Python, окна для редактирования исходного текста модуля, а также окна, в которые выводится результат выполнения программы (рис. П2.1).

При вызове программы IDLE первой появляется интерактивная оболочка ***Python Shell***. Работа с ней мало чем отличается от работы с Python в интерактивном режиме.

Начать редактировать новый файл можно, выбрав в меню **File** команду **New window**. В появившемся окне можно набирать текст программы. После набора текста не забудьте его сохранить: **File | Save**. IDLE откажется запустить программу на исполнение, если текст программы не сохранен.

Теперь можно запустить программу функциональной клавишей <F5> (или **Edit | Run module**). В ответ появляется новое окно ***Output*** с результатами выполнения программы.

Если вы допустили синтаксическую ошибку, в окне ***Output*** будет напечатана трассировочная информация, в том числе строки вида:

```
File "имя-файла", line номер-строки
```

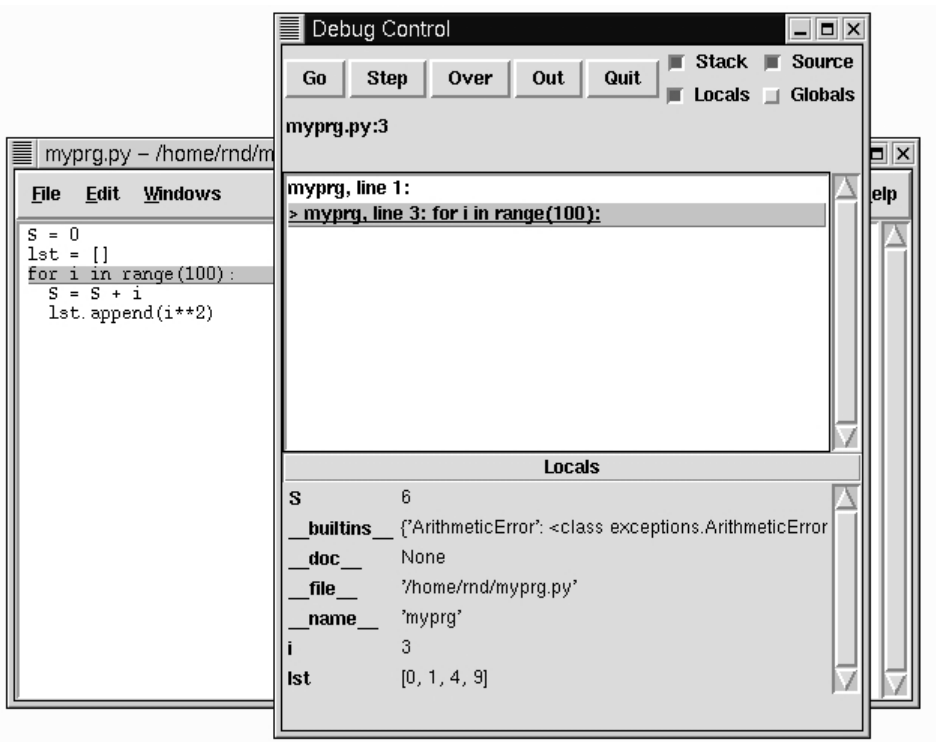


Рис. П2.2. Отладчик в процессе работы

Вы можете перейти к проблемному месту, выбрав правой кнопкой мыши позицию контекстного меню **Goto file/line**. В исходном тексте строка с ошибкой будет выделена (если модуль, в котором произошла ошибка, не был загружен, это будет сделано).

Запустить программу можно и с использованием *отладчика*, для чего можно нажать <Ctrl>+<F5> (или выбрать из меню **Edit** команду **Run debugger**). На рис. П2.2 представлен отладчик в процессе работы. Для пошагового исполнения программы можно использовать кнопки **Step** или **Over**. При использовании первой отладчик будет заходить во все функции, встречающиеся на пути, при использовании второй отладчик не будет углубляться в определения функций. Кнопка **Out** выводит отладчик из текущей функции, выполняя оставшиеся в функции команды "за кадром". Кнопка **Quit** завершает работу с отладчиком.

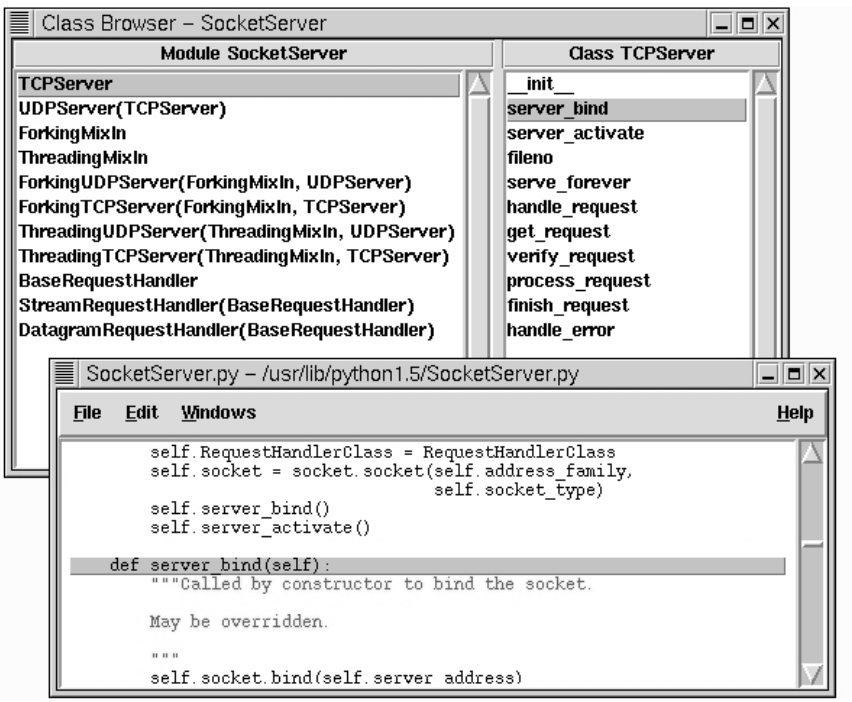


Рис. П2.3. Инструмент для просмотра классов модуля

В состав IDLE входит просмотрщик классов. Попробуйте открыть модуль (**File | Open module...**), в котором имеется нетривиальная структура классов, например `SocketServer`. В окне редактора открылся текст модуля, и теперь можно посмотреть на его структуру классов, выбрав команду **Class browser** в меню **File**. Щелкните мышью на интересующем вас классе (например,

`TCTPServer`), а затем на интересующем методе этого класса (например, `server_bind`). В результате будет выделена первая строка с описанием нужного метода, что и показано на рис. П2.3.

Аналогично работает и **Path browser** (инструмент для просмотра путей поиска модулей) из меню **File**: он позволяет углубиться в список путей поиска модулей, выбрать конкретный модуль, конкретный класс и метод этого класса. Эти два инструмента существенно облегчают ориентацию в сотнях модулей, классов и тысячах методов, которые скрываются в этих модулях.

В табл. П2.1 приведены переводы команд меню **Edit** оболочки IDLE.

Таблица П2.1. *Некоторые команды меню* ***Edit*** *оболочки IDLE*

Команда меню	Описание
Undo	Откатка назад последнего изменения в текущем окне (возможно до 1000 откаток)
Redo	Откатка откатки
Cut	Вырезает выделенный текст, помещая его в буфер обмена
Copy	Копирует выделенный текст в буфер обмена
Paste	Вставляет содержимое буфера обмена в текст сразу за точкой вставки
Select All	Выделяет весть текст в текущем окне
Find...	Открывает диалоговое окно для поиска
Find again	Повторяет предыдущую команду поиска
Find selection	Ищет строку, заданную выделением
Find in Files...	Открывает диалоговое окно для поиска файлов
Replace...	Открывает диалоговое окно для поиска-замены
Go to line	Спрашивает номер строки и переходит к заданной строке
Indent region	Сдвигает выделенные строки кода на четыре пробела вправо, увеличивая отступ
Dedent region	То же, влево, уменьшая отступ
Comment out region	Делает выделенные строки комментарием
Uncomment region	Снимает признак комментария с группы выделенных строк
Tabify region	Превращает начальные пробелы в символы табуляции
Untabify region	Превращает все табуляции в пробелы

Таблица П2.1 *(окончание)*

Команда меню	Описание
Expand word	Завершает ввод слова по образцу другого слова в текущем окне. Повторение команды дает другой вариант автозавершения. Полезно запомнить комбинацию "горячих" клавиш, соответствующую данному пункту меню
Format Paragraph	Переформатирует абзац
Run module	Импортирует или перезагружает текущий модуль (выполняет программу)
Debug module	То же, но под управлением отладчика

При работе с IDLE существуют некоторые ограничения по используемым средствам. IDLE написан с помощью Tkinter, и потому из него нельзя запускать приложения, также написанные на Tkinter. Другое ограничение касается использования потоков управления. Они тоже замысловатым образом могут помешать работе IDLE. И еще: к сожалению, IDLE, входящий в состав Python 2.0 и выше, не позволяет использовать кириллицу в модулях и интерактивной оболочке. Поэтому, если вам действительно нужен IDLE (например, для учебных целей), лучше воспользоваться версией Python 1.5.2. Проблемы с кириллицей в IDLE (и Tkinter) возникли из-за перехода нижележащего Tcl/Tk на Unicode, к чему соответствующие модули Python оказались не готовы. На момент написания книги эта досадная ошибка еще не была исправлена. Профессиональные разработчики Python практически не используют IDLE в качестве интерактивной среды разработки, предпочитая другие решения (текстовый редактор Emacs, Vi, а также среду от фирмы ActiveState Tool Corp. и т. п.).

Приложение 3

Описание компакт-диска



К книге прилагается компакт-диск, который прежде всего нужен для того, чтобы:

- обеспечить читателя всем необходимым для начала работы с Python;
- дать возможность тем, кто не имеет соединения с Internet, получить минимальный набор необходимых средств разработчика;
- предоставить читателю электронные копии примеров, использованных в книге.

Материалы на компакт-диске взяты из сети Internet, свободно распространяются, могут использоваться в коммерческих разработках, их применение не ущемляет ничьих авторских прав. Все программное обеспечение, как правило, представлено как в форме исходных текстов, так и в скомпилированном, готовом к исполнению виде. К каждому распространяемому продукту приложена лицензия, дающая право использовать его при соблюдении некоторых простых условий.

Этот компакт-диск является дополнением к книге и потому наиболее полезен только в сочетании с ней. Для просмотра содержимого можно использовать любой Web-браузер. Начать просмотр можно с файла index.html в корне диска.

В каталоге `examples`, в подкаталогах `/koi`, `/mac` и `/win` содержатся практически все примеры, встречающиеся в книге в виде листингов. Содержимое каталогов отличается кодировкой и символами конца строки:

- `/koi` — кодировка koi8-r; признак конца строки `0x0A`;
- `/mac` — MacCyrillic; `0x0D`;
- `/win` — Windows-1251; `0x0A`, `0x0D`.

В каталоге `/small_examples` собраны файлы с текстами (почти) всех примеров из книги. Подкаталоги `/koi`, `/mac` и `/win` имеют то же значение, что и в `/examples`.

В каталоге `/install` находятся инструкции по установке интерпретатора Python.

В файле /links/index.html приведены ссылки на некоторые полезные ресурсы, доступные в сети Internet.

Каталог `/progs` содержит программное обеспечение: интерпретаторы языка Python для различных платформ, утилиты, GNU компилятор языка C для Windows (в подкаталоге `/mingw`) и другое свободно распространяемое программное обеспечение.

В файле index.html приведен полный список программного обеспечения, собранного на диске, а по приведенным там ссылкам можно добраться до аннотаций программ и кратких инструкций по установке.

Примечание

Лицензии на программное обеспечение находятся в архивных файлах, вместе с программным обеспечением. Почти ко всем программам имеются и архивы с исходными кодами (source code).

Программы были проверены на Windows 98, MacOS 7.5 и Linux RedHat 6.2. Конечно, сложно гарантировать, что все они будут успешно компилироваться и работать в любой системе. Если программа "не хочет" компилироваться на вашей платформе, убедитесь, что:

- используемый компилятор достаточно стандартный или предусмотрен разработчиками этой программы (почитайте файлы INSTALL и README). Все приведенные программы поддерживают компилятор GNU C;
- компилятору доступны заголовочные файлы от используемых библиотек, а компоновщику — сами эти библиотеки. Возможно, вам потребуется предварительно доустановить требуемые библиотеки.

Некоторые проблемы могут возникнуть и при установке программ из бинарных дистрибутивов. Обычно они связаны с тем, что отсутствуют требуемые для установки версии библиотек, других модулей и программ.

Предупреждение

Все файлы предоставлены "как есть", и ни составитель компакт-диска, ни авторы представленного на нем программного обеспечения, ни издательство не могут нести ответственности за результаты его использования (или неработоспособность отдельных программ на вашей платформе).

Список литературы

1. Бизли Д. М. Язык программирования Python. Справочник. — ДиаСофт, 2000.
2. Буч Г. Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992.
3. Мещеряков М. Инсталляция и использование Linux. — СПб.: БХВ-Петербург, 2001.
4. Россум ван Г., Дрейк Ф. Л. Дж., Откидач Д. С. и др. Язык программирования Python. — СПб.: АНО "Институт логики" — Невский Диалект, 2001.
5. Тассел ван Д. Стиль, разработка, эффективность, отладка и испытание программ. — М.: Мир, 1995.
6. Уэзерелл Ч. Этюды для программистов. — М.: Мир, 1982.
7. Филд А., Харрисон П. Функциональное программирование. — М.: Мир, 1993.
8. Beazley David, Van Rossum Guido. Python: Essential Reference. New Riders Publishing, 1999.
9. Brown Martin C. Python: The Complete Reference. McGraw-Hill Professional Publishing, 2001.
10. Chun Wesley J. Core Python Programming. Prentice Hall PTR, 2000.
11. Gauld Alan. Learn to Program Using Python: A Tutorial for Hobbyists, Self-Starters, and Those Who Want to Learn the Art of Programming. Addison-Wesley Pub CoAddison-Wesley Pub Co, 2001.
12. Grayson John E. Python and Tkinter Programming. Manning Publications Company, 1999.
13. Hammond Mark, Robinson Andy. Python Programming on Win32. O'Reilly, 2000.
14. Jones Christopher A., Fred L. Drake. Python & XML. O'Reilly & Associates, 2001.

15. Van Laningham Ivan. Teach Yourself Python in 24 Hours. Sams, 2000.
16. Latteier Amos, Pelletier Michel. The Zope Book. New Riders Publishing, 2001.
17. Lundh Frederik. Python Standard Library. O'Reilly & Associates, 2001.
18. Lutz Mark. Programming Python (2nd Edition). O'Reilly & Associates, 2001.
19. Lutz Mark, Ascher David. Learning Python. O'Reilly, 1999.

Предметный указатель

A

ASCII 23, 197
  таблица 270

C

CGI (Common Gateway Interface, интерфейс общего шлюза) 470
CRC (Cyclic Redundancy Check, циклический избыточный контроль) 556

D

DB API 686
DOM (Document Object Model, объектная модель документа) 542
DOM-дерево 543
DTML (Document Template Markup Language, язык разметки шаблонов документов) 700, 704

F

Free Software Foundation (Фонд свободного программного обеспечения) 138
Freeware (свободное программное обеспечение) 138

G

GPL (GNU Public License, публичная лицензия GNU) 138

H

HTML (Hypertext Markup Language, язык разметки гипертекста) 517

I

I-курсор 592

J

Jython 715
JPython 715

L

locale 626

M

MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения почты Internet) 508

N

Numeric Python (NumPy, численный Python) 658

O

Open Source Software (программное обеспечение с открытым исходным кодом) 138

P

Pack 122
PEP (Python Enhancement Proposal, предложение по расширению языка Python) 140
PSF (Python Software Foundation, Фонд программного обеспечения Python) 138

R

RFC 140
RFC-821 487
RFC-822 505
RFC-1321 555
RFC-1489 370
RFC-1725 489
RFC-1869 487
RFC-2396 481
RFC-2616 481

S

SAX (Simple API for XML, простой программный интерфейс для XML) 527
SGML (Standard Generalized Markup Language, стандартный обобщенный язык разметки) 517
SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты) 487
SWIG (Simplified Wrapper and Interface Generator, упрощенный упаковщик и генератор интерфейса) 142

T

Tcl/Tk язык Tcl и его графический интерфейс Tk 567
Tkinter модуль Python, оболочка Tcl/Tk 567

U

Unicode 366
  таблица русских букв 271
URI (Uniform Resource Identifier, универсальный идентификатор ресурса) 478
URL (Uniform Resource Locator, универсальный указатель местонахождения ресурса) 478

W, X, Z

WWW (World Wide Web, всемирная паутина) 478
XML (Extensible Markup Language, расширяемый язык разметки) 525
Zope-сценарий 709

А

Абстрагирование 97
Автозавершение 460
Авторизация 702
Агрегация 102
Анализ объектно-ориентированный (ООА) 96
Аргумент 170
  значение по умолчанию 254
  именованный 205
  позиционный 205
  специальный 254
  фактический 171
  формальный 171
  функции 171
Ассоциативность операций 191
Атом 195
Атрибут класса или объекта 98
Аутентификация 702

Б

Базисная линия 601
Базовый класс 101
Базы данных 686
Байт-код 565
Бинарная операция 190
Блок кода 302
Брат (sibling) 543
Буфер обмена 575

В

Виджет 117
  выравнивание текста 577
  границы 577
  дочерний 118
  кнопка (Button) 588
  меню (Menu) 595
  переключатель (Radiobutton) 590
  подчиненный 612
  поле ввода (Entry) 592
  рисунок (Canvas) 605
  создание 570
  текст (Text) 596
  флажок (Checkbutton) 590
  цвет 577, 581
  шрифт 579
  этикетка (Label) 587
Входное имя (login) 335
Выбор 29
Выделение среза (slicing) 205
Вызов
  функции 171
  функции или объекта 205
Выражение 190
  знакомство с 14
  индексное 205
  первичное 204
Вычитание 207

Г

Генератор простой 319
Главный модуль 79
Графический интерфейс пользователя (GUI Graphical User Interface) 117
Графический интерфейс Tk 567
Группы по интересам (SIG Special Interest Group) 144

Д

Декларативный оператор 253
Декомпозиция на модули 77
Деление на ноль 207
Дерево виджетов 117
Дескриптор:
  файловый 397
  события 572
Деструктор 177
Детерминант (определитель) матрицы 672
Дети 543
Дизайн объектно-ориентированный (ООД) 96
Дизассемблер 565
Дизъюнкция 29
Дистрибутив 636
  бинарный 636
  с исходным кодом 636

Длинное целое 18
Дочерний узел (child node) 543

Е

Единичная матрица 662

З

Заглушка 80
Замена подстрок 353
  множественная 73
  с использованием регулярных выражений 360
Замок 432
Запись в экспоненциальном формате 17
Захват файла (file lock) 452
Значение:
  (состояние) объекта 97
  по умолчанию 5, 63

И

И:
  побитовое 217
  логическое 212
Идентификатор 195
  исключения 248
Идиома 317
Иерархия 99
ИЛИ:
  исключающее побитовое 217
  логическое 212
  побитовое 217
Имя 16, 195
  глобальное 238
  локальное 238
Индексное выражение 205
Индексные узлы (inode) 428
Индивидуальность объекта 97
Инкапсуляция 99
Интернационализация 626
Интерпретатор 11
Интерфейс модулей 82
Исключение 292
  ArithmeticError 295
  AssertionError 295
  AttributeError 295
  EnvironmentError 295
  Exception 293
  FloatingPointError 296
  ImportError 296
  IndentationError 296
  IndexError 296
  IOError 297
  KeyboardInterrupt 297
  KeyError 297
  LookupError 298
  MemoryError 298
  NameError 298
  NotImplementedError 298
  OSError 299
  OverflowError 299
  RuntimeError 300
  StandardError 293
  SyntaxError 300
  SystemError 300
  SystemExit 293
  TabError 300
  TypeError 301
  UnboundLocalError 301
  UnicodeError 301
  ValueError 302
  WindowsError 302
  ZeroDivisionError 302
Истина 212
Истинностное значение 212
Итератор 318
Итерация 31

К

Класс 151, 98, 176
  экземпляр 177
Кодек (кодировщик-декодировщик) 367
Кодировка 366
Кодовый объект 188
Комментарий 147
Коммуникационный домен 495
Композиция 61
Константа 197
  строковая 197

Конструктор 98, 177
Контейнер 102
Конъюнкция 29
Копирование:
  глубокое 330
  поверхностное 330
  массивов 662
Корневой узел (root node) 542
Кортеж (tuple) 36, 162
  синтаксис 201
Культурная среда (locale) 626

Л

Лексикографический порядок 215
Летнее время (DST Daylight Saving Time) 381
Литерал:
  строковый 197
    Unicode 198
    длинный 197
    короткий 197
    необрабатываемый 198
  числовой 200
Логические операции (boolean operators) 212
Ложь 212
Локализация 626
Лямбда-выражение 219

М

Максимальное целое 327
Мантисса 17
Массив 389, 658
  ассоциативный 167
Менеджер расположения 569, 612
Меню:
  верхнего уровня 595
  всплывающие (popup) 595
  каскадные (cascade) 595
  ниспадающие (pulldown) 595
Метод 174
  __abs__() 229
  __add__() 228
  __and__() 230
  __call__() 235
  __cmp__ 223
  __coerce__() 230
  __complex__() 230
  __del__ 221
  __delattr__() 228
  __delitem__() 234
  __delslice__() 234
  __div__() 228
  __divmod__() 228
  __eq__() 224
  __float__() 230
  __ge__() 224
  __getattr__() 226
  __getitem__() 232
  __getslice__() 234
  __gt__() 224
  __hash__() 224
  __hex__() 231
  __iadd__() 229
  __iand__() 230
  __idiv__() 229
  __ilshift__() 230
  __imod__() 229
  __imul__() 229
  __init__ 220
  __int__() 230
  __invert__() 230
  __ior__() 230
  __ipow__() 229
  __irshift__() 230
  __isub__() 229
  __ixor__() 230
  __le__() 224
  __len__() 232
  __long__() 230
  __lshift__() 230
  __lt__() 224
  __mod__() 228
  __mul__() 228
  __ne__() 224
  __neg__() 229
  __nonzero__() 225
  __oct__() 231
  __or__() 230
  __pos__() 229
  __pow__() 228

(окончание рубрики см. на стр. 742)

Метод (*окончание*):
__radd__() 228
__rand__() 230
__rcmp__ 223
__rdiv__() 228
__rdivmod__() 228
__repr__ 222
__rlshift__() 230
__rmod__() 228
__rmul__() 228
__ror__() 230
__rpow__() 228
__rrshift__() 230
__rshift__() 230
__rsub__() 228
__rxor__() 230
__setattr__() 227
__setitem__() 234
__setslice__() 234
__str__ 222
__sub__() 228
__xor__() 230
класса 98
Многозадачность (multitasking) 406
Многопоточная (multithreading) программа 432
Множественное наследование 101
Модуль 77, 308
__builtin__ 350
__main__ 350
_winreg 621
anydbm 443
array 389
atexit 329
base64 503
binascii 504
binhex 504
calendar 387
cgi 470
cmath 378
codecs 367
colorsys 553
compileall 564
copy 330
cPickle 338
crypt 558
cStringIO 364
difflib 371
dircache 414
dis 565
errno 414
fcntl 452
gc 333
getopt 416
getpass 335
gettext 631
glob 418
grp 452
gzip 444
htmllib 521
httplib 483
Image 675
imghdr 554
imp 287
IOCTL 452
keyword 564
LinearAlgebra 672
locale 627
mailbox 515
math 376
md5 555
mimetools 508
MimeWriter 511
mmap 374
msvcrt 620
multifile 513
MySQLdb 691
Numeric 658
operator 335
os 394
os.path 411
pdb 463
pickle 338
popen2 419
poplib 489
pprint 340
profile 465
pwd 451
py_compile 564
pydoc 322
PyGreSQL 695
quopri 502
random 379
re 356
readline 460
repr 341

resource 455
rexec 559
rfc822 505
rlcompleter 462
rotor 557
select 421
sgmllib 517
sha 556
shelve 343
shutil 420
signal 423
smtplib 487
sndhdr 554
socket 495
stat 427
string 350
StringIO 364
struct 390
sys 326
telnetlib 493
tempfile 430
termios 457
thread 440
threading 432
time 381
tkFont 581
Tkinter 567
traceback 344
turtle 618
types 346
urllib 478
urlparse 481
UserDict 349
UserList 349
UserString 348
uu 502
wave 551
whichdb 444
whrandom 380
winsound 625
xml.parsers.expat 539
zipfile 448
zlib 446
главный 79
интерфейс 82
расширения 651
таблица символов 181
Мультимедиа 550
Мусор 152

Н

Начертание шрифта 579
НЕ:
  логическое 212
  побитовое 217
Нетерминальные символы 7
Нисходящее проектирование 79

О

Область действия имен:
  вложенная 306
Оболочка IDLE 728
Обработчик:
  исключений 248, 292
  события 117, 572
Объект 96, 151, 177
  время жизни 178
  значение (value) 151
  изменяемость (mutability) 151
  изменяемый (mutable) 151
  индивидуальность 97
  индивидуальность (identity) 151
  кодовый 287
  неизменяемый (immutable) 151
  тип (type) 151
  тип (поведение) объекта 97
  устойчивость 114
  файловый 183
  файлоподобный 183
  хэш-значение 170
Объект-срез 187
Объект-тип 182
Объявление типа документа (DTD Document Type Definition) 526
Ожидание событий 570
Окно 117
Операнд 190
Оператор:
  assert 252
  break 246
  class 256

(окончание рубрики см. на стр. 744)

Оператор (*окончание*):
continue 246
def 253
del 240
exec 263
for 244
global 253
import 260
pass 247
print 238
raise 251
return 247
try 248
while 243
выбора 29, 241
выражение 236
присваивания 237
простой (simple) 236
составной (compound) 236
условия 241
Операции сравнения 214
больше 214
больше или равно 214
меньше 214
меньше или равно 214
не принадлежит 214
не равно 214
принадлежит 214
равно 214
тождественно равно 214
тождественно не равно 214
Операция 190
not 212
форматирования 209
Определение функции 171
Оптимизация программ 312
Остаток от деления 208
Отладка 463
Отладчик 463, 730
Отображение файлов в памяти (memory mapping) 374
Отступ 149

П

Пакет 310
xml.dom 543
xml.sax 527
Папка в Zope 704
Параметр 170
Патч (заплатка) 371
Первичное выражение 204
Побитовая инверсия 217
Повторение последовательности: 207
Подпакет (sub-package) 310
Подпрограмма 21
Политика безопасности в Zope 703
Порядок 17
вычислений 193
Постановка задачи 80
Поток событий 570
Потоки управления (threads) 431
Потомок 542
Права в Zope 702
Правила округления 211
Предок 542
Представление в виде строки 203
Приглашение ко вводу 12
Приобретение (acquisition) объектов в Zope 711
Проблемы совместимости кода 308
Программа 19
ввод выражений 146
интерактивный (диалоговый) ввод 145
событийно-управляемая (event-driven) 117
файловый ввод 145
Программирование 19
модульное 77
объектно-ориентированное (ООП) 96
процедурное 60
функциональное 60
Проектирование (дизайн) 79
Пространство имен 303
локальное 304
глобальное 304
модуля 181
Профайлер 312, 463
Процесс 406
Псевдослучайное число 33, 379

P

Размер (кегль) шрифта 579
Размерность массива 658
Рамка 117
Расширенное присваивание 238
Ребенок 543
Регистр букв 7
Регулярное выражение 42, 104, 355
  специальные последовательности 357
Результат вызова функции 171
Рекурсия:
  максимальная глубина 327
Родитель 542
Роль 220
  в Zope 702

C

Сборка мусора (garbage collection) 152
Сдвиг:
  битов влево 217
  битов вправо 217
Семафор 432
Семейство шрифта 579
Сериализация объектов 338
Сетевые протоколы 468
Сжатие (компрессия) данных 446
Сигнал 423
Символ:
  # 148
  пробельный 148
  табуляция 146, 149
  специальный 197
  управляющий 197
Система публикации объектов 704
Словарь 47
  ключ (индекс) 167
  синтаксис 203
Сложение 207
Случайные числа 379
Событие 117
Совместимость кода 308
Сокет 495
Сообщение 99
  заголовочная часть 505
  поля заголовочной части (header fields) 505
  тело (body) 505
Сортировка:
  строк с учетом кодировки 627
  элементов массива 670
  элементов списка 163
Спецификатор преобразования (операция %) 209
Списковое включение (list comprehension) 202
Список 36, 163
  синтаксис 202
Стандартные системные ошибки 414
Стек протоколов (protocol stack) 469
Стиль программирования 314
Строка 158
  логическая 146
  представление в виде строки 203
  программы пустая 148
  физическая 146
Строка-разделитель (в MIME-сообщении) 513
СУБД 686
Сценарий настройки (setup script) 640
  в Zope 709

T

Таблица истинности 213
Табуляция:
  символ 146
Тело цикла 31, 243
Тип:
  Ellipsis 154
  None 154
  ufunc (Numeric) 664
  Unicode-строка 157
  xrange 157
  буфер 157
  длинное целое (long) 155
  класс 176
  комплексное число 155
  кортеж 157
  метод 174

(окончание рубрики см. на стр. 746)

Тип (*окончание*):
модуль 181
последовательность 157
с поддержкой вызова (callable) 170
словарь (dictionary) 167
соответствие 167
список 157
строка 157
функция 171
целое число (int) 155
число с плавающей запятой (точкой) 155
элементов массива 659
Точка вставки 592
Транзакция 688
Трассировочный объект 189
Тупик (deadlock) 434

У

Умножение 207
Умолчание 5, 63
Унарная операция 190
Универсальное координируемое время (UTC Universal Time Coordinated) 381
Устойчивость объекта 114
Устойчивые (persistent) объекты 343
Утверждение 252
Утечка памяти 179
Утилиты Distutils 636

Ф

Файл 54
текстовый 146
Файловый:
дескриптор 397
объект 183
Файлоподобный объект 183
Факториал 67
Флаг 218
Фокус ввода с клавиатуры 575
Форма:
Бэкуса-Наура 7
массива 659
Фрагмент кода 236
Фрактал 619
Фрейм исполнения (execution frame) 302
Фрейм-объект 189
Функция:
__import__() 286
abs() 268
apply() 273
buffer() 292
callable() 278
chr() 270
cmp() 270
coerce() 267
compile() 287
complex 266
dir() 282
divmod() 268
eval() 283
execfile() 283
filter() 275
float 266
getattr() 290
globals() 280
hasattr() 290
hash() 278
hex() 270
id() 278
input() 288
int 266
intern() 282
isinstance() 279
issubclass() 279
len() 270
list 266
locals() 280
long 266
map() 274
max() 272
md5.new 556
min() 272
oct() 270
open() 289
ord() 269
pow() 268
range() 272
raw_input() 288

reduce() 275
reload() 284
repr 266
round() 269
setattr() 290
slice() 291
str 266
tuple 266
type() 265, 280
unichr() 271
unicode() 271
vars() 280
xrange() 272
zip() 277
ввода/вывода 288
в математике 60
высшего порядка 73, 173
рекурсивная 67
чистая 60

Х

Хэш 167
Хэш-значение 170
Хэш-метод 224
Хэш-функция 224, 278

Ц

Целостность данных 695
Цикл 31
for 31, 244
while 243 31, 243
с предусловием 243
событий:
главный 570
Циклическая ссылка 179

Э

Экземпляр класса (class instance) 151, 177
Элемент:
взятие по индексу 205
Эмуляция 220

Я

Язык программирования Tcl 567